

Meta-theory of first-order predicate logic

Stefan Berghofer

April 29, 2009

Abstract

We present a formalization of parts of Melvin Fitting’s book “First-Order Logic and Automated Theorem Proving” [1]. The formalization covers the syntax of first-order logic, its semantics, the model existence theorem, a natural deduction proof calculus together with a proof of correctness and completeness, as well as the Löwenheim-Skolem theorem.

Contents

1	Miscellaneous Utilities	2
2	Terms and formulae	2
2.1	Closed terms and formulae	3
2.2	Substitution	3
2.3	Parameters	5
3	Semantics	7
4	Proof calculus	9
5	Correctness	12
6	Completeness	12
6.1	Consistent sets	13
6.2	Closure under subsets	17
6.3	Finite character	19
6.4	Enumerating datatypes	25
6.4.1	Enumerating pairs of natural numbers	25
6.4.2	Enumerating trees	27
6.4.3	Enumerating lists	27
6.4.4	Enumerating terms	28
6.5	Extension to maximal consistent sets	30
6.6	Hintikka sets and Herbrand models	35

6.7	Model existence theorem	40
6.8	Completeness for Natural Deduction	41

7 Löwenheim-Skolem theorem 46

1 Miscellaneous Utilities

Rules for manipulating goals where both the premises and the conclusion contain conjunctions of similar structure.

theorem *conjE'*: $P \wedge Q \implies (P \implies P') \implies (Q \implies Q') \implies P' \wedge Q'$
by *iprover*

theorem *conjE''*: $(\forall x. P x \longrightarrow Q x \wedge R x) \implies$
 $((\forall x. P x \longrightarrow Q x) \implies Q') \implies ((\forall x. P x \longrightarrow R x) \implies R') \implies Q' \wedge R'$
by *iprover*

Some facts about (in)finite sets

theorem [*simp*]: $\neg A \cap B = B - A$ **by** *blast*

theorem *Compl-UNIV-eq*: $\neg A = UNIV - A$ **by** *fast*

theorem *infinite-nonempty*: $\neg \text{finite } A \implies \exists x. x \in A$
by (*subgoal-tac A \neq {}*) *fast+*

declare *Diff-infinite-finite* [*simp*]

2 Terms and formulae

The datatypes of terms and formulae in *de Bruijn notation* are defined as follows:

datatype *'a term* = *Var nat* | *App 'a 'a term list*

datatype (*'a, 'b*) *form* =
FF
| *TT*
| *Pred 'b 'a term list*
| *And ('a, 'b) form ('a, 'b) form*
| *Or ('a, 'b) form ('a, 'b) form*
| *Impl ('a, 'b) form ('a, 'b) form*
| *Neg ('a, 'b) form*
| *Forall ('a, 'b) form*
| *Exists ('a, 'b) form*

We use *'a* and *'b* to denote the type of *function symbols* and *predicate symbols*, respectively. In applications *App a ts* and predicates *Pred a ts*, the

length of ts is considered to be a part of the function or predicate name, so $App\ a\ [t]$ and $App\ a\ [t,u]$ refer to different functions.

2.1 Closed terms and formulae

Many of the results proved in the following sections are restricted to closed terms and formulae. We call a term or formula *closed at level i* , if it only contains “loose” bound variables with indices smaller than i .

primrec

$closedt :: nat \Rightarrow 'a\ term \Rightarrow bool$
and $closedts :: nat \Rightarrow 'a\ term\ list \Rightarrow bool$

where

$closedt\ m\ (Var\ n) = (n < m)$
 $| closedt\ m\ (App\ a\ ts) = closedts\ m\ ts$
 $| closedts\ m\ [] = True$
 $| closedts\ m\ (t \# ts) = (closedt\ m\ t \wedge closedts\ m\ ts)$

primrec

$closed :: nat \Rightarrow ('a, 'b)\ form \Rightarrow bool$

where

$closed\ m\ FF = True$
 $| closed\ m\ TT = True$
 $| closed\ m\ (Pred\ b\ ts) = closedts\ m\ ts$
 $| closed\ m\ (And\ p\ q) = (closed\ m\ p \wedge closed\ m\ q)$
 $| closed\ m\ (Or\ p\ q) = (closed\ m\ p \wedge closed\ m\ q)$
 $| closed\ m\ (Impl\ p\ q) = (closed\ m\ p \wedge closed\ m\ q)$
 $| closed\ m\ (Neg\ p) = closed\ m\ p$
 $| closed\ m\ (Forall\ p) = closed\ (Suc\ m)\ p$
 $| closed\ m\ (Exists\ p) = closed\ (Suc\ m)\ p$

theorem *closedt-mono*: **assumes** $le: i \leq j$

shows $closedt\ i\ (t::'a\ term) \implies closedt\ j\ t$

and $closedts\ i\ (ts::'a\ term\ list) \implies closedts\ j\ ts$ **using** le

by (*induct t and ts simp-all*)

2.2 Substitution

We now define substitution functions for terms and formulae. When performing substitutions under quantifiers, we need to *lift* the terms to be substituted for variables, in order for the “loose” bound variables to point to the right position.

primrec

$substt :: 'a\ term \Rightarrow 'a\ term \Rightarrow nat \Rightarrow 'a\ term\ (-['/-] [300, 0, 0] 300)$
and $substts :: 'a\ term\ list \Rightarrow 'a\ term \Rightarrow nat \Rightarrow 'a\ term\ list\ (-['/-] [300, 0, 0] 300)$

where

$(Var\ i)[s/k] = (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i)$

| $(App\ a\ ts)[s/k] = App\ a\ (ts[s/k])$
| $[][s/k] = []$
| $(t\ \#\ ts)[s/k] = t[s/k]\ \#\ ts[s/k]$

primrec

liftt :: 'a term \Rightarrow 'a term
and *liftts* :: 'a term list \Rightarrow 'a term list

where

liftt (Var *i*) = Var (Suc *i*)
| *liftt* (App *a* *ts*) = App *a* (*liftts* *ts*)
| *liftts* [] = []
| *liftts* (*t* # *ts*) = *liftt* *t* # *liftts* *ts*

primrec

subst :: ('a, 'b) form \Rightarrow 'a term \Rightarrow nat \Rightarrow ('a, 'b) form (-['/-] [300, 0, 0] 300)

where

FF[*s/k*] = *FF*
| *TT*[*s/k*] = *TT*
| (*Pred* *b* *ts*)[*s/k*] = *Pred* *b* (*ts*[*s/k*])
| (*And* *p* *q*)[*s/k*] = *And* (*p*[*s/k*]) (*q*[*s/k*])
| (*Or* *p* *q*)[*s/k*] = *Or* (*p*[*s/k*]) (*q*[*s/k*])
| (*Impl* *p* *q*)[*s/k*] = *Impl* (*p*[*s/k*]) (*q*[*s/k*])
| (*Neg* *p*)[*s/k*] = *Neg* (*p*[*s/k*])
| (*Forall* *p*)[*s/k*] = *Forall* (*p*[*liftt* *s*/Suc *k*])
| (*Exists* *p*)[*s/k*] = *Exists* (*p*[*liftt* *s*/Suc *k*])

theorem *lift-closed* [*simp*]:

closedt 0 (*t*::'a term) \Longrightarrow *closedt* 0 (*liftt* *t*)
closedts 0 (*ts*::'a term list) \Longrightarrow *closedts* 0 (*liftts* *ts*)
by (*induct* *t* **and** *ts*) *simp-all*

theorem *subst-closedt* [*simp*]: **assumes** *u*: *closedt* 0 *u*

shows *closedt* (Suc *i*) *t* \Longrightarrow *closedt* *i* (*t*[*u*/*i*])
and *closedts* (Suc *i*) *ts* \Longrightarrow *closedts* *i* (*ts*[*u*/*i*]) **using** *u*
apply (*induct* *t* **and** *ts*)
apply *simp-all*
apply (*rule* *impI*)
apply (*rule* *closedt-mono*(1) [of 0])
apply *simp+*
done

theorem *subst-closed* [*simp*]:

closedt 0 *t* \Longrightarrow *closed* (Suc *i*) *p* \Longrightarrow *closed* *i* (*p*[*t*/*i*])
by (*induct* *p* *arbitrary*: *i* *t*) *simp-all*

theorem *subst-size* [*simp*]: *size* (*subst* *p* *t* *i*) = *size* *p*

by (*induct* *p* *arbitrary*: *i* *t*) *simp-all*

2.3 Parameters

The introduction rule *ForallI* for the universal quantifier, as well as the elimination rule *ExistsE* for the existential quantifier introduced in §4 require the quantified variable to be replaced by a “fresh” parameter. Fitting’s solution is to use a new nullary function symbol for this purpose. To express that a function symbol is “fresh”, we introduce functions for collecting all function symbols occurring in a term or formula.

primrec

$paramst :: 'a \text{ term} \Rightarrow 'a \text{ set}$
and $paramsts :: 'a \text{ term list} \Rightarrow 'a \text{ set}$
where
 $paramst (Var\ n) = \{\}$
 $| paramst (App\ a\ ts) = \{a\} \cup paramsts\ ts$
 $| paramsts\ [] = \{\}$
 $| paramsts\ (t\ \#\ ts) = (paramst\ t \cup paramsts\ ts)$

primrec

$params :: ('a, 'b) \text{ form} \Rightarrow 'a \text{ set}$
where
 $params\ FF = \{\}$
 $| params\ TT = \{\}$
 $| params\ (Pred\ b\ ts) = paramsts\ ts$
 $| params\ (And\ p\ q) = params\ p \cup params\ q$
 $| params\ (Or\ p\ q) = params\ p \cup params\ q$
 $| params\ (Impl\ p\ q) = params\ p \cup params\ q$
 $| params\ (Neg\ p) = params\ p$
 $| params\ (Forall\ p) = params\ p$
 $| params\ (Exists\ p) = params\ p$

We also define parameter substitution functions on terms and formulae that apply a function f to all function symbols.

primrec

$psubstf :: ('a \Rightarrow 'c) \Rightarrow 'a \text{ term} \Rightarrow 'c \text{ term}$
and $psubstfs :: ('a \Rightarrow 'c) \Rightarrow 'a \text{ term list} \Rightarrow 'c \text{ term list}$
where
 $psubstf\ f\ (Var\ i) = Var\ i$
 $| psubstf\ f\ (App\ x\ ts) = App\ (f\ x)\ (psubstfs\ f\ ts)$
 $| psubstfs\ f\ [] = []$
 $| psubstfs\ f\ (t\ \#\ ts) = psubstf\ f\ t\ \#\ psubstfs\ f\ ts$

primrec

$psubst :: ('a \Rightarrow 'c) \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('c, 'b) \text{ form}$
where
 $psubst\ f\ FF = FF$
 $| psubst\ f\ TT = TT$
 $| psubst\ f\ (Pred\ b\ ts) = Pred\ b\ (psubstfs\ f\ ts)$
 $| psubst\ f\ (And\ p\ q) = And\ (psubst\ f\ p)\ (psubst\ f\ q)$

$| \text{psubst } f \text{ (Or } p \text{ } q) = \text{Or } (\text{psubst } f \text{ } p) \text{ (psubst } f \text{ } q)$
 $| \text{psubst } f \text{ (Impl } p \text{ } q) = \text{Impl } (\text{psubst } f \text{ } p) \text{ (psubst } f \text{ } q)$
 $| \text{psubst } f \text{ (Neg } p) = \text{Neg } (\text{psubst } f \text{ } p)$
 $| \text{psubst } f \text{ (Forall } p) = \text{Forall } (\text{psubst } f \text{ } p)$
 $| \text{psubst } f \text{ (Exists } p) = \text{Exists } (\text{psubst } f \text{ } p)$

theorem *psubstt-closed* [simp]:
 $\text{closedt } i \text{ (psubstt } f \text{ } t) = \text{closedt } i \text{ } t$
 $\text{closedts } i \text{ (psubstts } f \text{ } ts) = \text{closedts } i \text{ } ts$
by (induct *t* and *ts*) *simp-all*

theorem *psubst-closed* [simp]:
 $\text{closed } i \text{ (psubst } f \text{ } p) = \text{closed } i \text{ } p$
by (induct *p* arbitrary: *i*) *simp-all*

theorem *psubstt-subst* [simp]:
 $\text{psubstt } f \text{ (substt } t \text{ } u \text{ } i) = \text{substt } (\text{psubstt } f \text{ } t) \text{ (psubstt } f \text{ } u) \text{ } i$
 $\text{psubstts } f \text{ (substts } ts \text{ } u \text{ } i) = \text{substts } (\text{psubstts } f \text{ } ts) \text{ (psubstt } f \text{ } u) \text{ } i$
by (induct *t* and *ts*) *simp-all*

theorem *psubstt-lift* [simp]:
 $\text{psubstt } f \text{ (liftt } t) = \text{liftt } (\text{psubstt } f \text{ } t)$
 $\text{psubstts } f \text{ (liftts } ts) = \text{liftts } (\text{psubstts } f \text{ } ts)$
by (induct *t* and *ts*) *simp-all*

theorem *psubst-subst* [simp]:
 $\text{psubst } f \text{ (subst } P \text{ } t \text{ } i) = \text{subst } (\text{psubst } f \text{ } P) \text{ (psubstt } f \text{ } t) \text{ } i$
by (induct *P* arbitrary: *i* *t*) *simp-all*

theorem *psubstt-upd* [simp]:
 $x \notin \text{paramst } (t::'a \text{ term}) \implies \text{psubstt } (f(x:=y)) \text{ } t = \text{psubstt } f \text{ } t$
 $x \notin \text{paramsts } (ts::'a \text{ term list}) \implies \text{psubstts } (f(x:=y)) \text{ } ts = \text{psubstts } f \text{ } ts$
by (induct *t* and *ts*) (*auto split add: sum.split*)

theorem *psubst-upd* [simp]: $x \notin \text{params } P \implies \text{psubst } (f(x:=y)) \text{ } P = \text{psubst } f \text{ } P$
by (induct *P*) (*simp-all del: fun-upd-apply*)

theorem *psubstt-id* [simp]: $\text{psubstt } (\%x. x) \text{ (} t::'a \text{ term)} = t$
 $\text{psubstts } (\%x. x) \text{ (} ts::'a \text{ term list)} = ts$
by (induct *t* and *ts*) *simp-all*

theorem *psubst-id* [simp]: $\text{psubst } (\%x. x) = (\%p. p)$
apply (*rule ext*)
apply (*induct-tac p*)
apply *simp-all*
done

theorem *psubstt-image* [simp]:
 $\text{paramst } (\text{psubstt } f \text{ } t) = f \text{ ' paramst } t$

$paramsts (psubstts f ts) = f \text{ ' } paramsts ts$
by (*induct t and ts*) (*simp-all add: image-Un*)

theorem *psubst-image* [*simp*]: $params (psubst f p) = f \text{ ' } params p$
by (*induct p*) (*simp-all add: image-Un*)

3 Semantics

In this section, we define evaluation functions for terms and formulae. Evaluation is performed relative to an environment mapping indices of variables to values. We also introduce a function, denoted by $e\langle i:a \rangle$, for inserting a value a at position i into the environment. All values of variables with indices less than i are left untouched by this operation, whereas the values of variables with indices greater or equal than i are shifted one position up.

definition

$shift :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a \text{ (-\langle -: \rangle) [90, 0, 0] 91)}$ **where**
 $e\langle i:a \rangle = (\lambda j. \text{if } j < i \text{ then } e\ j \text{ else if } j = i \text{ then } a \text{ else } e\ (j - 1))$

lemma *shift-eq* [*simp*]: $i = j \Longrightarrow (e\langle i:T \rangle) j = T$
by (*simp add: shift-def*)

lemma *shift-gt* [*simp*]: $j < i \Longrightarrow (e\langle i:T \rangle) j = e\ j$
by (*simp add: shift-def*)

lemma *shift-lt* [*simp*]: $i < j \Longrightarrow (e\langle i:T \rangle) j = e\ (j - 1)$
by (*simp add: shift-def*)

lemma *shift-commute* [*simp*]: $e\langle i:U \rangle \langle 0:T \rangle = e\langle 0:T \rangle \langle Suc\ i:U \rangle$
apply (*rule ext*)
apply (*case-tac x*)
apply *simp*
apply (*case-tac nat*)
apply (*simp-all add: shift-def*)
done

primrec

$evalt :: (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c\ list \Rightarrow 'c) \Rightarrow 'a\ term \Rightarrow 'c$
and $evalts :: (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c\ list \Rightarrow 'c) \Rightarrow 'a\ term\ list \Rightarrow 'c\ list$

where

$evalt\ e\ f\ (Var\ n) = e\ n$
 $| evalt\ e\ f\ (App\ a\ ts) = f\ a\ (evalts\ e\ f\ ts)$
 $| evalts\ e\ f\ [] = []$
 $| evalts\ e\ f\ (t\ \#\ ts) = evalt\ e\ f\ t\ \# evalts\ e\ f\ ts$

primrec

$eval :: (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c\ list \Rightarrow 'c) \Rightarrow$
 $('b \Rightarrow 'c\ list \Rightarrow bool) \Rightarrow ('a, 'b)\ form \Rightarrow bool$

where

$eval\ e\ f\ g\ FF = False$
 $| eval\ e\ f\ g\ TT = True$
 $| eval\ e\ f\ g\ (Pred\ a\ ts) = g\ a\ (evalts\ e\ f\ ts)$
 $| eval\ e\ f\ g\ (And\ p\ q) = ((eval\ e\ f\ g\ p) \wedge (eval\ e\ f\ g\ q))$
 $| eval\ e\ f\ g\ (Or\ p\ q) = ((eval\ e\ f\ g\ p) \vee (eval\ e\ f\ g\ q))$
 $| eval\ e\ f\ g\ (Impl\ p\ q) = ((eval\ e\ f\ g\ p) \longrightarrow (eval\ e\ f\ g\ q))$
 $| eval\ e\ f\ g\ (Neg\ p) = (\neg (eval\ e\ f\ g\ p))$
 $| eval\ e\ f\ g\ (Forall\ p) = (\forall z. eval\ (e\langle 0:z \rangle)\ f\ g\ p)$
 $| eval\ e\ f\ g\ (Exists\ p) = (\exists z. eval\ (e\langle 0:z \rangle)\ f\ g\ p)$

We write $e, f, g, ps \models p$ to mean that the formula p is a semantic consequence of the list of formulae p with respect to an environment e and interpretations f and g for function and predicate symbols, respectively.

definition

$model :: (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c\ list \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c\ list \Rightarrow bool) \Rightarrow$
 $('a, 'b) form\ list \Rightarrow ('a, 'b) form \Rightarrow bool \quad (\neg, \neg, \neg, \neg \models - [50, 50] 50) \quad \mathbf{where}$
 $(e, f, g, ps \models p) = (list\ all\ (eval\ e\ f\ g)\ ps \longrightarrow eval\ e\ f\ g\ p)$

The following substitution lemmas relate substitution and evaluation functions:

theorem *subst-lemma'* [simp]:

$evalt\ e\ f\ (subst\ t\ u\ i) = evalt\ (e\langle i:evalt\ e\ f\ u \rangle)\ f\ t$
 $evalts\ e\ f\ (substts\ ts\ u\ i) = evalts\ (e\langle i:evalt\ e\ f\ u \rangle)\ f\ ts$
by (induct t and ts) simp-all

theorem *lift-lemma* [simp]:

$evalt\ (e\langle 0:z \rangle)\ f\ (liftt\ t) = evalt\ e\ f\ t$
 $evalts\ (e\langle 0:z \rangle)\ f\ (liftts\ ts) = evalts\ e\ f\ ts$
by (induct t and ts) simp-all

theorem *subst-lemma* [simp]:

$\bigwedge e\ i\ t. eval\ e\ f\ g\ (subst\ a\ t\ i) = eval\ (e\langle i:evalt\ e\ f\ t \rangle)\ f\ g\ a$
by (induct a) simp-all

theorem *upd-lemma'* [simp]:

$n \notin paramst\ t \Longrightarrow evalt\ e\ (f(n:=x))\ t = evalt\ e\ f\ t$
 $n \notin paramsts\ ts \Longrightarrow evalts\ e\ (f(n:=x))\ ts = evalts\ e\ f\ ts$
by (induct t and ts) auto

theorem *upd-lemma* [simp]:

$n \notin params\ p \Longrightarrow eval\ e\ (f(n:=x))\ g\ p = eval\ e\ f\ g\ p$
by (induct p arbitrary: e) simp-all

theorem *list-upd-lemma* [simp]: $list\ all\ (\lambda p. n \notin params\ p)\ G \Longrightarrow$

$list\ all\ (eval\ e\ (f(n:=x))\ g)\ G = list\ all\ (eval\ e\ f\ g)\ G$
by (induct G) simp-all

In order to test the evaluation function defined above, we apply it to an

example:

theorem *ex-all-commute-eval*:

eval e f g (Impl (Exists (Forall (Pred p [Var 1, Var 0])))
(Forall (Exists (Pred p [Var 0, Var 1]))))

apply *simp*

Simplification yields the following proof state:

1. $(\exists z. \forall za. g p [z, za]) \longrightarrow (\forall z. \exists za. g p [za, z])$

This is easily proved using intuitionistic logic:

apply *iprover*

done

4 Proof calculus

We now introduce a natural deduction proof calculus for first order logic. The derivability judgement $G \vdash a$ is defined as an inductive predicate.

inductive

deriv :: ('a, 'b) form list \Rightarrow ('a, 'b) form \Rightarrow bool (- \vdash - [50,50] 50)

where

Assum: $a \text{ mem } G \Longrightarrow G \vdash a$

| *TTI*: $G \vdash TT$

| *FFE*: $G \vdash FF \Longrightarrow G \vdash a$

| *NegI*: $a \# G \vdash FF \Longrightarrow G \vdash \text{Neg } a$

| *NegE*: $G \vdash \text{Neg } a \Longrightarrow G \vdash a \Longrightarrow G \vdash FF$

| *Class*: $\text{Neg } a \# G \vdash FF \Longrightarrow G \vdash a$

| *AndI*: $G \vdash a \Longrightarrow G \vdash b \Longrightarrow G \vdash \text{And } a b$

| *AndE1*: $G \vdash \text{And } a b \Longrightarrow G \vdash a$

| *AndE2*: $G \vdash \text{And } a b \Longrightarrow G \vdash b$

| *OrI1*: $G \vdash a \Longrightarrow G \vdash \text{Or } a b$

| *OrI2*: $G \vdash b \Longrightarrow G \vdash \text{Or } a b$

| *OrE*: $G \vdash \text{Or } a b \Longrightarrow a \# G \vdash c \Longrightarrow b \# G \vdash c \Longrightarrow G \vdash c$

| *ImplI*: $a \# G \vdash b \Longrightarrow G \vdash \text{Impl } a b$

| *ImplE*: $G \vdash \text{Impl } a b \Longrightarrow G \vdash a \Longrightarrow G \vdash b$

| *ForallI*: $G \vdash a[\text{App } n \ []/0] \Longrightarrow \text{list-all } (\lambda p. n \notin \text{params } p) G \Longrightarrow$
 $n \notin \text{params } a \Longrightarrow G \vdash \text{Forall } a$

| *ForallE*: $G \vdash \text{Forall } a \Longrightarrow G \vdash a[t/0]$

| *ExistsI*: $G \vdash a[t/0] \Longrightarrow G \vdash \text{Exists } a$

| *ExistsE*: $G \vdash \text{Exists } a \Longrightarrow a[\text{App } n \ []/0] \# G \vdash b \Longrightarrow$

$\text{list-all } (\lambda p. n \notin \text{params } p) G \Longrightarrow n \notin \text{params } a \Longrightarrow n \notin \text{params } b \Longrightarrow G \vdash b$

The following derived inference rules are sometimes useful in applications.

theorem *cut*: $G \vdash A \Longrightarrow A \# G \vdash B \Longrightarrow G \vdash B$

by (*rule ImplE*) (*rule ImplI*)

theorem *cut'*: $A \# G \vdash B \Longrightarrow G \vdash A \Longrightarrow G \vdash B$

by (rule ImplE) (rule ImplI)

theorem *Class'*: $Neg\ A \ \# \ G \vdash\ A \implies\ G \vdash\ A$
 apply (rule Class)
 apply (rule-tac $A=A$ in cut')
 apply (rule-tac $a=A$ in NegE)
 apply (rule Assum)
 apply simp
 apply (rule Assum)
 apply simp
 apply assumption
 done

theorem *ForallE'*: $G \vdash\ Forall\ a \implies\ subst\ a\ t\ 0 \ \# \ G \vdash\ B \implies\ G \vdash\ B$
 by (rule cut) (rule ForallE)

As an example, we show that the excluded middle, a commutation property for existential and universal quantifiers, the drinker principle, as well as Peirce's law are derivable in the calculus given above.

theorem *tnd*: $\ [] \vdash\ Or\ (Pred\ p\ [])\ (Neg\ (Pred\ p\ []))$
 apply (rule Class)
 apply (rule NegE)
 apply (rule Assum, simp)
 apply (rule OrI2)
 apply (rule NegI)
 apply (rule NegE)
 apply (rule Assum, simp)
 apply (rule OrI1)
 apply (rule Assum, simp)
 done

theorem *ex-all-commute*:
 $(\ []::(nat, 'b)\ form\ list) \vdash\ Impl\ (Exists\ (Forall\ (Pred\ p\ [Var\ 1,\ Var\ 0])))$
 $(Forall\ (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1])))$
 apply (rule ImplI)
 apply (rule-tac $n=0$ in ForallI)
 prefer 2
 apply simp
 prefer 2
 apply simp
 apply simp
 apply (rule-tac $n=1$ and $a=Forall\ (Pred\ p\ [Var\ 1,\ Var\ 0])$ in ExistsE)
 apply (rule Assum, simp)
 prefer 2
 apply simp
 prefer 2
 apply simp
 prefer 2
 apply simp

```

apply simp
apply (rule-tac  $t = \text{App } 1 \ []$  in ExistsI)
apply simp
apply (rule-tac  $t = \text{App } 0 \ []$  and  $a = \text{Pred } p \ [\text{App } (\text{Suc } 0) \ [], \text{Var } 0]$  in ForallE')
apply (rule Assum, simp)
apply (rule Assum, simp)
done

theorem drinker: ( $[\ ] :: (\text{nat}, 'b) \text{ form list}$ )  $\vdash$ 
  Exists (Impl (Pred  $P \ [\text{Var } 0]$ ) (Forall (Pred  $P \ [\text{Var } 0]$ )))
apply (rule Class')
apply (rule-tac  $t = \text{Var } 0$  in ExistsI)
apply simp
apply (rule ImplI)
apply (rule-tac  $n = 0$  in ForallI)
prefer 2
apply simp
prefer 2
apply simp
apply simp
apply (rule Class)
apply (rule-tac  $a = \text{Exists}$  (Impl (Pred  $P \ [\text{Var } 0]$ ) (Forall (Pred  $P \ [\text{Var } 0]$ ))) in
NegE)
apply (rule Assum, simp)
apply (rule-tac  $t = \text{App } 0 \ []$  in ExistsI)
apply simp
apply (rule ImplI)
apply (rule FFE)
apply (rule-tac  $a = \text{Pred } P \ [\text{App } 0 \ []]$  in NegE)
apply (rule Assum, simp)
apply (rule Assum, simp)
done

theorem peirce:
 $[\ ] \vdash \text{Impl} (\text{Impl} (\text{Impl} (\text{Pred } P \ [])) (\text{Pred } Q \ [])) (\text{Pred } P \ []) (\text{Pred } P \ [])$ 
apply (rule Class')
apply (rule ImplI)
apply (rule-tac  $a = \text{Impl} (\text{Pred } P \ []) (\text{Pred } Q \ [])$  in ImplE)
apply (rule Assum, simp)
apply (rule ImplI)
apply (rule FFE)
apply (rule-tac
   $a = \text{Impl} (\text{Impl} (\text{Impl} (\text{Pred } P \ []) (\text{Pred } Q \ [])) (\text{Pred } P \ [])) (\text{Pred } P \ [])$ 
in NegE)
apply (rule Assum, simp)
apply (rule ImplI)
apply (rule Assum, simp)
done

```

5 Correctness

The correctness of the proof calculus introduced in §4 can now be proved by induction on the derivation of $G \vdash p$, using the substitution rules proved in §3.

```

theorem correctness:  $G \vdash p \implies \forall e f g. e, f, g, G \models p$ 
  apply (unfold model-def)
  apply (erule deriv.induct)
  apply (rule allI impI)+
  apply (simp add: list-all-iff mem-iff)
  apply simp-all
  apply blast+
  apply (rule allI impI)+
  apply (erule-tac x=e in allE)
  apply (erule-tac x=f(n:=λx. z) in allE)
  apply (simp del: fun-upd-apply add: fun-upd-same)
  apply iprover
  apply (rule allI impI)+
  apply (erule allE, erule allE, erule allE, erule impE, assumption)
  apply (erule exE)
  apply (erule-tac x=e in allE)
  apply (erule-tac x=f(n:=λx. z) in allE)
  apply (simp del: fun-upd-apply add: fun-upd-same)
done

```

6 Completeness

The goal of this section is to prove completeness of the natural deduction calculus introduced in §4. Before we start with the actual proof, it is useful to note that the following two formulations of completeness are equivalent:

1. All valid formulae are derivable, i.e. $ps \models p \implies ps \vdash p$
2. All consistent sets are satisfiable

The latter property is called the *model existence theorem*. To see why 2 implies 1, observe that $Neg p, ps \not\models FF$ implies that $Neg p, ps$ is consistent, which, by the model existence theorem, implies that $Neg p, ps$ has a model, which in turn implies that $ps \not\models p$. By contraposition, it therefore follows from $ps \models p$ that $Neg p, ps \vdash FF$, which allows us to deduce $ps \vdash p$ using rule *Class*.

In most textbooks on logic, a set S of formulae is called *consistent*, if no contradiction can be derived from S using a *specific proof calculus*, i.e. $S \not\models FF$. Rather than defining consistency relative to a *specific* calculus, Fitting uses the more general approach of describing properties that all consistent sets must have (see §6.1).

The key idea behind the proof of the model existence theorem is to extend a consistent set to one that is *maximal* (see §6.5). In order to do this, we use the fact that the set of formulae is enumerable (see §6.4), which allows us to form a sequence $\phi_0, \phi_1, \phi_2, \dots$ containing all formulae. We can then construct a sequence S_i of consistent sets as follows:

$$S_0 = S$$

$$S_{i+1} = \begin{cases} S_i \cup \{\phi_i\} & \text{if } S_i \cup \{\phi_i\} \text{ consistent} \\ S_i & \text{otherwise} \end{cases}$$

To obtain a maximal consistent set, we form the union $\bigcup_i S_i$ of these sets. To ensure that this union is still consistent, additional closure (see §6.2) and finiteness (see §6.3) properties are needed. It can be shown that a maximal consistent set is a *Hintikka set* (see §6.6). Hintikka sets are satisfiable in *Herbrand* models, where closed terms coincide with their interpretation.

6.1 Consistent sets

In this section, we describe an abstract criterion for consistent sets. A set of sets of formulae is called a *consistency property*, if the following holds:

definition

$$\begin{aligned} \text{consistency} &:: ('a, 'b) \text{ form set set} \Rightarrow \text{bool} \text{ \textbf{where}} \\ \text{consistency } C &= (\forall S. S \in C \longrightarrow \\ &(\forall p \text{ ts. } \neg (Pred \text{ p ts} \in S \wedge Neg (Pred \text{ p ts}) \in S)) \wedge \\ &FF \notin S \wedge Neg TT \notin S \wedge \\ &(\forall Z. Neg (Neg Z) \in S \longrightarrow S \cup \{Z\} \in C) \wedge \\ &(\forall A B. And A B \in S \longrightarrow S \cup \{A, B\} \in C) \wedge \\ &(\forall A B. Neg (Or A B) \in S \longrightarrow S \cup \{Neg A, Neg B\} \in C) \wedge \\ &(\forall A B. Or A B \in S \longrightarrow S \cup \{A\} \in C \vee S \cup \{B\} \in C) \wedge \\ &(\forall A B. Neg (And A B) \in S \longrightarrow S \cup \{Neg A\} \in C \vee S \cup \{Neg B\} \in C) \wedge \\ &(\forall A B. Impl A B \in S \longrightarrow S \cup \{Neg A\} \in C \vee S \cup \{B\} \in C) \wedge \\ &(\forall A B. Neg (Impl A B) \in S \longrightarrow S \cup \{A, Neg B\} \in C) \wedge \\ &(\forall P t. closedt 0 t \longrightarrow Forall P \in S \longrightarrow S \cup \{P[t/0]\} \in C) \wedge \\ &(\forall P t. closedt 0 t \longrightarrow Neg (Exists P) \in S \longrightarrow S \cup \{Neg (P[t/0])\} \in C) \wedge \\ &(\forall P. Exists P \in S \longrightarrow (\exists x. S \cup \{P[App x []/0]\} \in C)) \wedge \\ &(\forall P. Neg (Forall P) \in S \longrightarrow (\exists x. S \cup \{Neg (P[App x []/0])\} \in C))) \end{aligned}$$

In §6.3, we will show how to extend a consistency property to one that is of *finite character*. However, the above definition of a consistency property cannot be used for this, since there is a problem with the treatment of formulae of the form *Exists P* and *Neg (Forall P)*. Fitting therefore suggests to define an *alternative consistency property* as follows:

definition

$$\begin{aligned} \text{alt-consistency} &:: ('a, 'b) \text{ form set set} \Rightarrow \text{bool} \text{ \textbf{where}} \\ \text{alt-consistency } C &= (\forall S. S \in C \longrightarrow \\ &(\forall p \text{ ts. } \neg (Pred \text{ p ts} \in S \wedge Neg (Pred \text{ p ts}) \in S)) \wedge \end{aligned}$$

$$\begin{aligned}
& FF \notin S \wedge \text{Neg } TT \notin S \wedge \\
& (\forall Z. \text{Neg } (\text{Neg } Z) \in S \longrightarrow S \cup \{Z\} \in C) \wedge \\
& (\forall A B. \text{And } A B \in S \longrightarrow S \cup \{A, B\} \in C) \wedge \\
& (\forall A B. \text{Neg } (\text{Or } A B) \in S \longrightarrow S \cup \{\text{Neg } A, \text{Neg } B\} \in C) \wedge \\
& (\forall A B. \text{Or } A B \in S \longrightarrow S \cup \{A\} \in C \vee S \cup \{B\} \in C) \wedge \\
& (\forall A B. \text{Neg } (\text{And } A B) \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{\text{Neg } B\} \in C) \wedge \\
& (\forall A B. \text{Impl } A B \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{B\} \in C) \wedge \\
& (\forall A B. \text{Neg } (\text{Impl } A B) \in S \longrightarrow S \cup \{A, \text{Neg } B\} \in C) \wedge \\
& (\forall P t. \text{closedt } 0 t \longrightarrow \text{Forall } P \in S \longrightarrow S \cup \{P[t/0]\} \in C) \wedge \\
& (\forall P t. \text{closedt } 0 t \longrightarrow \text{Neg } (\text{Exists } P) \in S \longrightarrow S \cup \{\text{Neg } (P[t/0])\} \in C) \wedge \\
& (\forall P x. (\forall a \in S. x \notin \text{params } a) \longrightarrow \text{Exists } P \in S \longrightarrow \\
& \quad S \cup \{P[\text{App } x \ _ / 0]\} \in C) \wedge \\
& (\forall P x. (\forall a \in S. x \notin \text{params } a) \longrightarrow \text{Neg } (\text{Forall } P) \in S \longrightarrow \\
& \quad S \cup \{\text{Neg } (P[\text{App } x \ _ / 0])\} \in C)
\end{aligned}$$

Note that in the clauses for *Exists* P and *Neg* (*Forall* P), the first definition requires the existence of a parameter x with a certain property, whereas the second definition requires that all parameters x that are new for S have a certain property. A consistency property can easily be turned into an alternative consistency property by applying a suitable parameter substitution:

definition

mk-alt-consistency :: ('a, 'b) form set set \Rightarrow ('a, 'b) form set set **where**
mk-alt-consistency $C = \{S. \exists f. \text{psubst } f \ 'S \in C\}$

theorem *alt-consistency*:

consistency $C \Longrightarrow$ *alt-consistency* (*mk-alt-consistency* C)
apply (*simp add: consistency-def alt-consistency-def mk-alt-consistency-def*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule exE*)
apply (*erule allE impE*)
apply *assumption*
apply (*erule conjE'*)
apply (*rule allI impI notI*)
apply (*erule allE impE*)
apply (*rule image-eqI*)
prefer 2
apply *assumption*
apply (*rule psubst.simps [symmetric]*)
apply (*erule notE*)
apply (*rule image-eqI*)
prefer 2
apply (*rotate-tac -1*)
apply *assumption*
apply *simp*
apply (*erule conjE'*)
apply (*rule notI*)
apply (*erule notE*)
apply (*rule image-eqI*)

```

prefer 2
apply assumption
apply simp
apply (erule conjE')
apply (rule notI)
apply (erule notE)
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (erule exI)
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply (rule psubst.simps [symmetric])
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (rule conjI, (rule refl)+)
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply (rule psubst.simps [symmetric])
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption

```

```

apply simp
apply iprover
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply iprover
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply iprover
apply iprover
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE)
apply (erule-tac x=psubstt f t in allE)
apply (erule impE)
apply simp
apply (erule impE)
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (erule exI)
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE)
apply (erule-tac x=psubstt f t in allE)
apply (erule impE)
apply simp
apply (erule impE)
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (erule exI)
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)

```

```

prefer 2
apply assumption
apply simp
apply (erule exE)
apply (rule-tac x=f(x:=xa) in exI)
apply (frule bspec)
apply assumption
apply (simp cong add: image-cong)
apply (rule allI impI)+
apply (erule allE impE)+
apply (rule image-eqI)
prefer 2
apply assumption
apply simp
apply (erule exE)
apply (rule-tac x=f(x:=xa) in exI)
apply (frule bspec)
apply assumption
apply (simp cong add: image-cong)
done

```

```

theorem mk-alt-consistency-subset: C ⊆ mk-alt-consistency C
apply (unfold mk-alt-consistency-def)
apply (rule subsetI)
apply (rule CollectI)
apply (rule-tac x=%x. x in exI)
apply simp
done

```

6.2 Closure under subsets

We now show that a consistency property can be extended to one that is closed under subsets.

definition

close :: ('a, 'b) form set set ⇒ ('a, 'b) form set set **where**
close C = {S. ∃ S' ∈ C. S ⊆ S'}

definition

subset-closed :: 'a set set ⇒ bool **where**
subset-closed C = (∀ S' ∈ C. ∀ S. S ⊆ S' → S ∈ C)

theorem *close-consistency: consistency C ⇒ consistency (close C)*

```

apply (simp add: close-def consistency-def)
apply (rule allI)
apply (rule impI)
apply (erule bexE)
apply (erule allE impE)+
apply assumption
apply (erule conjE', blast)

```

apply (*erule conjE'*, *blast*)
apply (*erule conjE'*, *blast*)
apply (*erule conjE'*, *blast*)
apply (*erule conjE'*, *blast*)
apply (*erule conjE'*, *blast*)
apply (*erule conjE'*)
apply (*rule allI impI*)
apply (*erule allE impE*)
apply (*erule subsetD*)
apply *assumption*
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*rule-tac x=insert A x in bexI*)
apply *blast*
apply *assumption*
apply (*rule disjI2*)
apply (*rule-tac x=insert B x in bexI*)
apply *blast*
apply *assumption*
apply (*erule conjE'*)
apply (*rule allI impI*)
apply (*erule allE impE*)
apply (*erule subsetD*)
apply *assumption*
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*rule-tac x=insert (Neg A) x in bexI*)
apply *blast*
apply *assumption*
apply (*rule disjI2*)
apply (*rule-tac x=insert (Neg B) x in bexI*)
apply *blast*
apply *assumption*
apply (*erule conjE'*)
apply (*rule allI impI*)
apply (*erule allE impE*)
apply (*erule subsetD*)
apply *assumption*
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*rule-tac x=insert (Neg A) x in bexI*)
apply *blast*
apply *assumption*
apply (*rule disjI2*)
apply (*rule-tac x=insert B x in bexI*)
apply *blast*
apply *assumption*
apply (*erule conjE'*, *blast*)
apply (*erule conjE'*, *blast*)

```

apply (erule conjE', blast)
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (erule subsetD)
apply assumption
apply blast
apply (rule allI impI)+
apply (erule allE impE)+
apply (erule subsetD)
apply assumption
apply blast
done

```

theorem *close-closed: subset-closed* (close C)
by (unfold close-def subset-closed-def) blast

theorem *close-subset: $C \subseteq$ close C*
by (unfold close-def) blast

If a consistency property C is closed under subsets, so is the corresponding alternative consistency property:

theorem *mk-alt-consistency-closed:*
subset-closed $C \implies$ subset-closed (mk-alt-consistency C)
apply (unfold mk-alt-consistency-def subset-closed-def)
apply (rule ballI allI impI)+
apply (rule CollectI)
apply (erule CollectE)
apply (erule exE)
apply (subgoal-tac psubst $f' S \subseteq$ psubst $f' S'$)
apply blast+
done

6.3 Finite character

In this section, we show that an alternative consistency property can be extended to one of finite character. A set of sets C is said to be of finite character, provided that S is a member of C if and only if every subset of S is.

definition

finite-char :: 'a set set \Rightarrow bool **where**
finite-char $C = (\forall S. S \in C = (\forall S'. \text{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in C))$

definition

mk-finite-char :: 'a set set \Rightarrow 'a set set **where**
mk-finite-char $C = \{S. \forall S'. S' \subseteq S \longrightarrow \text{finite } S' \longrightarrow S' \in C\}$

theorem *finite-alt-consistency:*

alt-consistency C \implies *subset-closed C* \implies *alt-consistency (mk-finite-char C)*
apply (*unfold alt-consistency-def subset-closed-def mk-finite-char-def*)
apply (*rule allI impI*)
apply (*erule CollectE*)
apply (*erule conjE''*)
apply (*rule allI notI*)
apply (*rotate-tac I*)
apply (*erule-tac x={Pred p ts, Neg (Pred p ts)} in allE*)
apply *blast*
apply (*erule conjE''*)
apply (*rotate-tac I*)
apply (*erule-tac x={FF} in allE*)
apply *blast*
apply (*erule conjE''*)
apply (*rotate-tac I*)
apply (*erule-tac x={Neg TT} in allE*)
apply *blast*
apply (*erule conjE''*)
apply (*rule allI impI CollectI*)
apply (*rotate-tac I*)
apply (*erule-tac x=S' - {Z} \cup {Neg (Neg Z)}* in allE)
apply (*erule impE*)
apply *blast*
apply (*erule impE*)
apply (*simp (no-asm)*)
apply (*erule allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac x=Z in allE*)
apply (*erule impE*)
apply (*simp (no-asm)*)
apply (*erule-tac x=S' - {Z} \cup {Neg (Neg Z)} \cup {Z}* in bspec)
apply *assumption*
apply *blast*
apply (*erule conjE''*)
apply (*rule allI impI CollectI*)
apply (*rotate-tac I*)
apply (*erule-tac x=S' - {A, B} \cup {And A B}* in allE)
apply (*erule impE*)
apply *blast*
apply (*erule impE*)
apply (*simp (no-asm)*)
apply (*erule allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac x=A in allE*)
apply (*erule-tac x=B in allE*)
apply (*erule impE*)
apply (*simp (no-asm)*)

```

apply (drule-tac x=S' - {A, B} ∪ {And A B} ∪ {A, B} in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (rotate-tac 1)
apply (erule-tac x=S' - {Neg A, Neg B} ∪ {Neg (Or A B)} in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=A in allE)
apply (erule-tac x=B in allE)
apply (erule impE)
apply (simp (no-asm))
apply (drule-tac x=S' - {Neg A, Neg B} ∪ {Neg (Or A B)} ∪ {Neg A, Neg B}
in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+
apply (rule ccontr)
apply (simp (no-asm-use))
apply (erule conjE exE)+
apply (rotate-tac 1)
apply (erule-tac x=(S' - {A}) ∪ (S'a - {B}) ∪ {Or A B} in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm-simp))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=A in allE)
apply (erule-tac x=B in allE)
apply (erule impE)
apply (simp (no-asm))
apply (erule disjE)
apply (drule-tac x=insert A (S' - {A}) ∪ (S'a - {B}) ∪ {Or A B} in bspec)
apply assumption
apply blast
apply (drule-tac x=insert B (S' - {A}) ∪ (S'a - {B}) ∪ {Or A B} in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+

```

```

apply (rule ccontr)
apply (simp (no-asm-use))
apply (erule conjE exE)+
apply (rotate-tac 1)
apply (erule-tac x=(S' - {Neg A}) ∪ (S'a - {Neg B}) ∪ {Neg (And A B)}) in
allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm-simp))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=A in allE)
apply (erule-tac x=B in allE)
apply (erule impE)
apply (simp (no-asm))
apply (erule disjE)
apply (drule-tac x=insert (Neg A) (S' - {Neg A}) ∪ (S'a - {Neg B}) ∪ {Neg
(And A B)}) in bspec)
apply assumption
apply blast
apply (drule-tac x=insert (Neg B) (S' - {Neg A}) ∪ (S'a - {Neg B}) ∪ {Neg
(And A B)}) in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+
apply (rule ccontr)
apply (simp (no-asm-use))
apply (erule conjE exE)+
apply (rotate-tac 1)
apply (erule-tac x=(S' - {Neg A}) ∪ (S'a - {B}) ∪ {Impl A B}) in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm-simp))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=A in allE)
apply (erule-tac x=B in allE)
apply (erule impE)
apply (simp (no-asm))
apply (erule disjE)
apply (drule-tac x=insert (Neg A) (S' - {Neg A}) ∪ (S'a - {B}) ∪ {Impl A
B}) in bspec)
apply assumption
apply blast

```

```

apply (drule-tac x=insert B (S' - {Neg A} ∪ (S'a - {B}) ∪ {Impl A B}) in
bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (rotate-tac 1)
apply (erule-tac x=S' - {A, Neg B} ∪ {Neg (Impl A B)} in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=A in allE)
apply (erule-tac x=B in allE)
apply (erule impE)
apply (simp (no-asm))
apply (drule-tac x=S' - {A, Neg B} ∪ {Neg (Impl A B)} ∪ {A, Neg B} in
bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (rotate-tac 1)
apply (erule-tac x=S' - {subst P t 0} ∪ {Forall P} in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm))
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule-tac x=P in allE)
apply (erule-tac x=t in allE)
apply (erule impE)
apply assumption
apply (drule-tac x=S' - {subst P t 0} ∪ {Forall P} ∪ {subst P t 0} in bspec)
apply blast
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (rotate-tac 1)
apply (erule-tac x=S' - {Neg (subst P t 0)} ∪ {Neg (Exists P)} in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply (simp (no-asm))

```

apply (*erule allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac x=P in allE*)
apply (*erule-tac x=t in allE*)
apply (*erule impE*)
apply *assumption*
apply (*drule-tac x=S' - {Neg (subst P t 0)} ∪ {Neg (Exists P)} ∪ {Neg (subst P t 0)}*) **in** *bspec*)
apply *blast*
apply *blast*
apply (*erule conjE''*)
apply (*rule allI impI CollectI*)
apply (*rotate-tac 1*)
apply (*erule-tac x=S' - {subst P (App x []) 0} ∪ {Exists P}*) **in** *allE*)
apply (*erule impE*)
apply *blast*
apply (*erule impE*)
apply (*simp (no-asm)*)
apply (*erule allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac x=P in allE*)
apply (*erule-tac x=x in allE*)
apply (*erule impE*)
apply *blast*
apply (*drule-tac x=S' - {subst P (App x []) 0} ∪ {Exists P} ∪ {subst P (App x []) 0}*) **in** *bspec*)
apply *blast*
apply *blast*
apply (*rule allI impI CollectI*)
apply (*rotate-tac 1*)
apply (*erule-tac x=S' - {Neg (subst P (App x []) 0)} ∪ {Neg (Forall P)}*) **in** *allE*)
apply (*erule impE*)
apply *blast*
apply (*erule impE*)
apply (*simp (no-asm)*)
apply (*erule allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac x=P in allE*)
apply (*erule-tac x=x in allE*)
apply (*erule impE*)
apply *blast*
apply (*drule-tac x=S' - {Neg (subst P (App x []) 0)} ∪ {Neg (Forall P)} ∪ {Neg (subst P (App x []) 0)}*) **in** *bspec*)
apply *blast*
apply *blast*

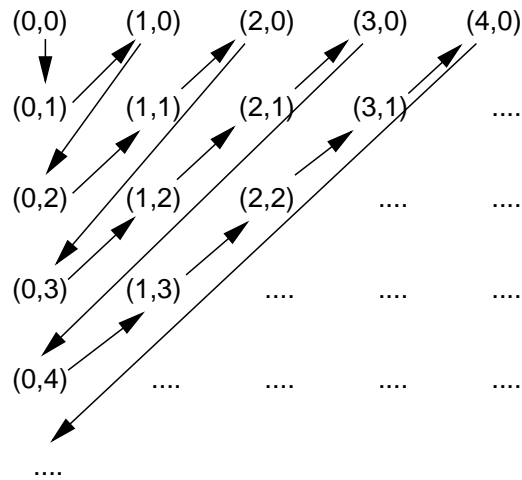


Figure 1: Cantor's method for enumerating sets of pairs

done

theorem *finite-char*: *finite-char* (*mk-finite-char* *C*)
by (*unfold finite-char-def mk-finite-char-def*) *blast*

theorem *finite-char-closed*: *finite-char* *C* \implies *subset-closed* *C*
apply (*unfold finite-char-def subset-closed-def*)
apply (*rule ballI allI impI*)
apply (*frule spec*)
apply (*erule iffD2*)
apply *blast*
done

theorem *finite-char-subset*: *subset-closed* *C* \implies $C \subseteq$ *mk-finite-char* *C*
by (*unfold mk-finite-char-def subset-closed-def*) *blast*

6.4 Enumerating datatypes

In the following section, we will show that elements of datatypes can be enumerated. This will be done by specifying functions that map natural numbers to elements of datatypes and vice versa.

6.4.1 Enumerating pairs of natural numbers

As a starting point, we show that pairs of natural numbers are enumerable. For this purpose, we use a method due to Cantor, which is illustrated in Figure 1. The function for mapping natural numbers to pairs of natural numbers can be characterized recursively as follows:

primrec

$diag :: nat \Rightarrow (nat \times nat)$

where

$diag\ 0 = (0, 0)$

| $diag\ (Suc\ n) =$

$(let\ (x, y) = diag\ n$

 in case y of

$0 \Rightarrow (0, Suc\ x)$

 | $Suc\ y \Rightarrow (Suc\ x, y)$)

theorem *diag-le1*: $fst\ (diag\ (Suc\ n)) < Suc\ n$

by (*induct n*) (*simp-all add: Let-def split-def split add: nat.split*)

theorem *diag-le2*: $snd\ (diag\ (Suc\ (Suc\ n))) < Suc\ (Suc\ n)$

apply (*induct n*)

apply (*simp-all add: Let-def split-def split add: nat.split nat.split-asm*)

apply (*rule impI*)

apply (*case-tac n*)

apply *simp*

apply *hypsubst*

apply (*rule diag-le1*)

done

theorem *diag-le3*: $fst\ (diag\ n) = Suc\ x \Longrightarrow snd\ (diag\ n) < n$

apply (*case-tac n*)

apply *simp*

apply (*case-tac nat*)

apply (*simp add: Let-def*)

apply *hypsubst*

apply (*rule diag-le2*)

done

theorem *diag-le4*: $fst\ (diag\ n) = Suc\ x \Longrightarrow x < n$

apply (*case-tac n*)

apply *simp*

apply (*case-tac nat*)

apply (*simp add: Let-def*)

apply *hypsubst*

apply (*drule sym*)

apply (*drule ord-eq-less-trans*)

apply (*rule diag-le1*)

apply *simp*

done

function

$undiaq :: nat \times nat \Rightarrow nat$

where

$undiaq\ (0, 0) = 0$

| $undiaq\ (0, Suc\ y) = Suc\ (undiaq\ (y, 0))$

| *undia*g (Suc x, y) = Suc (*undia*g (x, Suc y))
 by *pat-completeness auto*

termination

by (*relation measure* ($\lambda(x, y). ((x + y) * (x + y + 1)) \text{ div } 2 + x$)) *auto*

theorem *diag-undia*g [simp]: *diag* (*undia*g (x, y)) = (x, y)

by (*rule undia*g.induct) (*simp add: Let-def*)⁺

6.4.2 Enumerating trees

When writing enumeration functions for datatypes, it is useful to note that all datatypes are some kind of trees. In order to avoid re-inventing the wheel, we therefore write enumeration functions for trees once and for all. In applications, we then only have to write functions for converting between trees and concrete datatypes.

datatype *btree* = *Leaf nat* | *Branch btree btree*

function

diag-btree :: *nat* \Rightarrow *btree*

where

diag-btree n = (*case fst* (*diag* n) of
 0 \Rightarrow *Leaf* (*snd* (*diag* n))
 | *Suc* x \Rightarrow *Branch* (*diag-btree* x) (*diag-btree* (*snd* (*diag* n))))
 by *auto*

termination

by (*relation measure* ($\lambda x. x$)) (*auto intro: diag-le3 diag-le4*)

primrec

*undia*g-btree :: *btree* \Rightarrow *nat*

where

*undia*g-btree (*Leaf* n) = *undia*g (0, n)
 | *undia*g-btree (*Branch* t1 t2) =
*undia*g (*Suc* (*undia*g-btree t1), *undia*g-btree t2)

theorem *diag-undia*g-btree [simp]: *diag-btree* (*undia*g-btree t) = t

by (*induct* t) (*simp-all add: Let-def*)

declare *diag-btree.simps* [simp del] *undia*g-btree.simps [simp del]

6.4.3 Enumerating lists

fun

list-of-btree :: (*nat* \Rightarrow 'a) \Rightarrow *btree* \Rightarrow 'a *list*

where

list-of-btree f (*Leaf* x) = []
 | *list-of-btree* f (*Branch* (*Leaf* n) t) = f n # *list-of-btree* f t

primrec

$btree\text{-of-list} :: ('a \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow btree$

where

$btree\text{-of-list } f \ [] = Leaf\ 0$

$| btree\text{-of-list } f (x \# xs) = Branch (Leaf (f x)) (btree\text{-of-list } f xs)$

definition

$diag\text{-list} :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a\ list$ **where**

$diag\text{-list } f\ n = list\text{-of-btree } f (diag\text{-btree } n)$

definition

$undiaq\text{-list} :: ('a \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat$ **where**

$undiaq\text{-list } f\ xs = undiaq\text{-btree } (btree\text{-of-list } f\ xs)$

theorem $diag\text{-undiaq-list}$ [simp]:

$(\bigwedge x. d (u x) = x) \Longrightarrow diag\text{-list } d (undiaq\text{-list } u\ xs) = xs$

by (induct xs) (simp-all add: diag-list-def undiaq-list-def)

6.4.4 Enumerating terms**fun**

$term\text{-of-btree} :: (nat \Rightarrow 'a) \Rightarrow btree \Rightarrow 'a\ term$

and $term\text{-list-of-btree} :: (nat \Rightarrow 'a) \Rightarrow btree \Rightarrow 'a\ term\ list$

where

$term\text{-of-btree } f (Leaf\ m) = Var\ m$

$| term\text{-of-btree } f (Branch (Leaf\ m)\ t) =$

$App (f\ m) (term\text{-list-of-btree } f\ t)$

$| term\text{-list-of-btree } f (Leaf\ m) = []$

$| term\text{-list-of-btree } f (Branch\ t1\ t2) =$

$term\text{-of-btree } f\ t1 \# term\text{-list-of-btree } f\ t2$

primrec

$btree\text{-of-term} :: ('a \Rightarrow nat) \Rightarrow 'a\ term \Rightarrow btree$

and $btree\text{-of-term-list} :: ('a \Rightarrow nat) \Rightarrow 'a\ term\ list \Rightarrow btree$

where

$btree\text{-of-term } f (Var\ m) = Leaf\ m$

$| btree\text{-of-term } f (App\ m\ ts) = Branch (Leaf (f m)) (btree\text{-of-term-list } f\ ts)$

$| btree\text{-of-term-list } f \ [] = Leaf\ 0$

$| btree\text{-of-term-list } f (t \# ts) = Branch (btree\text{-of-term } f\ t) (btree\text{-of-term-list } f\ ts)$

theorem $term\text{-btree}$: **assumes** $du: \bigwedge x. d (u x) = x$

shows $term\text{-of-btree } d (btree\text{-of-term } u\ t) = t$

and $term\text{-list-of-btree } d (btree\text{-of-term-list } u\ ts) = ts$

by (induct t **and** ts) (simp-all add: du)

definition

$diag\text{-term} :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a\ term$ **where**

$diag\text{-term } f\ n = term\text{-of-btree } f (diag\text{-btree } n)$

definition

undia-term :: ('a ⇒ nat) ⇒ 'a term ⇒ nat **where**
undia-term f t = *undia*-btree (btree-of-term f t)

theorem *diag-undia-term* [*simp*]:

($\bigwedge x. d (u x) = x$) ⇒ *diag-term* d (*undia-term* u t) = t
by (*simp add: diag-term-def undia-term-def term-btree*)

fun

form-of-btree :: (nat ⇒ 'a) ⇒ (nat ⇒ 'b) ⇒ btree ⇒ ('a, 'b) form
where

form-of-btree f g (Leaf 0) = FF
| *form-of-btree* f g (Leaf (Suc 0)) = TT
| *form-of-btree* f g (Branch (Leaf 0) (Branch (Leaf m) (Leaf n))) =
 Pred (g m) (*diag-list* (*diag-term* f) n)
| *form-of-btree* f g (Branch (Leaf (Suc 0)) (Branch t1 t2)) =
 And (*form-of-btree* f g t1) (*form-of-btree* f g t2)
| *form-of-btree* f g (Branch (Leaf (Suc (Suc 0))) (Branch t1 t2)) =
 Or (*form-of-btree* f g t1) (*form-of-btree* f g t2)
| *form-of-btree* f g (Branch (Leaf (Suc (Suc (Suc 0)))) (Branch t1 t2)) =
 Impl (*form-of-btree* f g t1) (*form-of-btree* f g t2)
| *form-of-btree* f g (Branch (Leaf (Suc (Suc (Suc (Suc 0)))))) t =
 Neg (*form-of-btree* f g t)
| *form-of-btree* f g (Branch (Leaf (Suc (Suc (Suc (Suc (Suc 0)))))) t) =
 Forall (*form-of-btree* f g t)
| *form-of-btree* f g (Branch (Leaf (Suc (Suc (Suc (Suc (Suc (Suc 0)))))) t) =
 Exists (*form-of-btree* f g t)

primrec

btree-of-form :: ('a ⇒ nat) ⇒ ('b ⇒ nat) ⇒ ('a, 'b) form ⇒ btree
where

btree-of-form f g FF = Leaf 0
| *btree-of-form* f g TT = Leaf (Suc 0)
| *btree-of-form* f g (Pred b ts) = Branch (Leaf 0)
 (Branch (Leaf (g b)) (Leaf (*undia-list* (*undia-term* f) ts)))
| *btree-of-form* f g (And a b) = Branch (Leaf (Suc 0))
 (Branch (*btree-of-form* f g a) (*btree-of-form* f g b))
| *btree-of-form* f g (Or a b) = Branch (Leaf (Suc (Suc 0)))
 (Branch (*btree-of-form* f g a) (*btree-of-form* f g b))
| *btree-of-form* f g (Impl a b) = Branch (Leaf (Suc (Suc (Suc 0))))
 (Branch (*btree-of-form* f g a) (*btree-of-form* f g b))
| *btree-of-form* f g (Neg a) = Branch (Leaf (Suc (Suc (Suc (Suc 0))))))
 (*btree-of-form* f g a)
| *btree-of-form* f g (Forall a) = Branch (Leaf (Suc (Suc (Suc (Suc (Suc 0))))))
 (*btree-of-form* f g a)
| *btree-of-form* f g (Exists a) = Branch
 (Leaf (Suc (Suc (Suc (Suc (Suc (Suc 0))))))
 (*btree-of-form* f g a))

definition

$diag\text{-form} :: (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow nat \Rightarrow ('a, 'b) \text{ form}$ **where**
 $diag\text{-form } f \ g \ n = \text{form-of-btree } f \ g \ (\text{diag-btree } n)$

definition

$undia\text{-form} :: ('a \Rightarrow nat) \Rightarrow ('b \Rightarrow nat) \Rightarrow ('a, 'b) \text{ form} \Rightarrow nat$ **where**
 $undia\text{-form } f \ g \ x = \text{undia\text{-btree}} \ (\text{btree-of-form } f \ g \ x)$

theorem $diag\text{-undia\text{-form}}$ [simp]:

$(\bigwedge x. d \ (u \ x) = x) \Longrightarrow (\bigwedge x. d' \ (u' \ x) = x) \Longrightarrow$
 $diag\text{-form } d \ d' \ (\text{undia\text{-form } u \ u' \ f}) = f$
by ($\text{induct } f$) ($\text{simp-all add: } diag\text{-form-def undia\text{-form-def}$)

definition

$diag\text{-form}' :: nat \Rightarrow (nat, nat) \text{ form}$ **where**
 $diag\text{-form}' = \text{diag\text{-form}} \ (\lambda n. n) \ (\lambda n. n)$

definition

$undia\text{-form}' :: (nat, nat) \text{ form} \Rightarrow nat$ **where**
 $undia\text{-form}' = \text{undia\text{-form}} \ (\lambda n. n) \ (\lambda n. n)$

theorem $diag\text{-undia\text{-form}'$ [simp]: $diag\text{-form}' \ (\text{undia\text{-form}' } f) = f$

by ($\text{simp add: } diag\text{-form}'\text{-def undia\text{-form}'\text{-def}$)

6.5 Extension to maximal consistent sets

Given a set C of finite character, we show that the least upper bound of a chain of sets that are elements of C is again an element of C .

definition

$is\text{-chain} :: (nat \Rightarrow 'a \text{ set}) \Rightarrow bool$ **where**
 $is\text{-chain } f = (\forall n. f \ n \subseteq f \ (\text{Suc } n))$

theorem $is\text{-chainD}$: $is\text{-chain } f \Longrightarrow x \in f \ m \Longrightarrow x \in f \ (m + n)$

by ($\text{induct } n$) ($\text{fastsimp simp add: } is\text{-chain-def}$) $+$

theorem $is\text{-chainD}'$: $is\text{-chain } f \Longrightarrow x \in f \ m \Longrightarrow m \leq k \Longrightarrow x \in f \ k$

apply ($\text{subgoal-tac } \exists n. k = m + n$)

apply ($\text{erule } exE$)

apply simp

apply ($\text{erule-tac } n=n$ **in** $is\text{-chainD}$)

apply assumption

apply arith

done

theorem $chain\text{-index}$:

assumes ch : $is\text{-chain } f$ **and** fin : $\text{finite } F$

shows $F \subseteq (\bigcup n. f \ n) \Longrightarrow \exists n. F \subseteq f \ n$ **using** fin

apply ($\text{induct rule: } \text{finite-induct}$)

```

apply blast
apply (insert ch)
apply simp
apply (erule conjE exE)+
apply (rule-tac x=max n xa in exI)
apply (rule conjI)
apply (erule is-chainD')
apply assumption
apply (simp add: max-def)
apply (rule subsetI)
apply (erule-tac B=f n in subsetD)
apply assumption
apply (erule is-chainD')
apply assumption
apply (simp add: max-def)
done

```

theorem *chain-union-closed*:

```

finite-char C  $\implies$  is-chain f  $\implies$   $\forall n. f\ n \in C \implies (\bigcup n. f\ n) \in C$ 
apply (erule finite-char-closed)
apply (unfold finite-char-def subset-closed-def)
apply (erule spec)
apply (erule iffD2)
apply (rule allI impI)+
apply (erule chain-index)
apply blast+
done

```

We can now define a function *Extend* that extends a consistent set to a maximal consistent set. To this end, we first define an auxiliary function *extend* that produces the elements of an ascending chain of consistent sets.

primrec

```

dest-Neg :: ('a, 'b) form  $\Rightarrow$  ('a, 'b) form

```

where

```

dest-Neg (Neg p) = p

```

primrec

```

dest-Forall :: ('a, 'b) form  $\Rightarrow$  ('a, 'b) form

```

where

```

dest-Forall (Forall p) = p

```

primrec

```

dest-Exists :: ('a, 'b) form  $\Rightarrow$  ('a, 'b) form

```

where

```

dest-Exists (Exists p) = p

```

primrec

```

extend :: (nat, 'b) form set  $\Rightarrow$  (nat, 'b) form set set  $\Rightarrow$ 
  (nat  $\Rightarrow$  (nat, 'b) form)  $\Rightarrow$  nat  $\Rightarrow$  (nat, 'b) form set

```

where

```
extend S C f 0 = S
| extend S C f (Suc n) = (if extend S C f n ∪ {f n} ∈ C
  then
    (if (∃ p. f n = Exists p)
      then extend S C f n ∪ {f n} ∪ {subst (dest-Exists (f n))
        (App (SOME k. k ∉ (∪ p ∈ extend S C f n ∪ {f n}. params p)) [] ) 0}
      else if (∃ p. f n = Neg (Forall p))
        then extend S C f n ∪ {f n} ∪ {Neg (subst (dest-Forall (dest-Neg (f n)))
          (App (SOME k. k ∉ (∪ p ∈ extend S C f n ∪ {f n}. params p)) [] ) 0)}
        else extend S C f n ∪ {f n})
    else extend S C f n)
```

definition

```
Extend :: (nat, 'b) form set ⇒ (nat, 'b) form set set ⇒
  (nat ⇒ (nat, 'b) form) ⇒ (nat, 'b) form set where
Extend S C f = (∪ n. extend S C f n)
```

theorem *is-chain-extend*: *is-chain* (extend S C f)

by (simp add: is-chain-def) blast

theorem *finite-paramst* [simp]: *finite* (paramst (t :: 'a term))

finite (paramsts (ts :: 'a term list))

by (induct t **and** ts) (simp-all split add: sum.split)

theorem *finite-params* [simp]: *finite* (params p)

by (induct p) simp-all

theorem *finite-params-extend* [simp]:

\neg *finite* ($\bigcap p \in S. \neg$ params p) \implies \neg *finite* ($\bigcap p \in$ extend S C f n. \neg params p)

by (induct n) simp-all

theorem *extend-in-C*: *alt-consistency* C \implies

$S \in C \implies \neg$ *finite* (\neg ($\bigcup p \in S. \text{params } p$)) \implies extend S C f n $\in C$

apply (induct n)

apply simp-all

apply (rule conjI impI)+

apply (erule exE)+

apply simp

apply (simp add: alt-consistency-def)

apply (rule impI)+

apply (erule exE)

apply (erule-tac x=insert (f n) (extend S C f n) **in** allE)

apply (erule impE)

apply assumption

apply ((erule conjunct2)+,

erule-tac x=p **in** allE,

erule-tac x=SOME k.

k \notin ($\bigcup x \in$ extend S C f n \cup {f n}. params x) **in** allE)

```

apply (erule impE)
apply (subgoal-tac  $\neg$  finite ( $-\ (\bigcup x \in \text{extend } S \ C \ f \ n \cup \{f \ n\}. \text{params } x)$ ))
prefer 2
apply simp
apply (rule ballI)
apply (drule-tac  $A = - \ ?S$  in infinite-nonempty)
apply (erule exE)
apply (rule someI2)
apply (simp only: Compl-iff [symmetric])
apply fast
apply simp
apply (rule impI)+
apply (erule exE)
apply (simp add: alt-consistency-def)
apply (erule-tac  $x = \text{insert } (Exists \ p)$  (extend  $S \ C \ f \ n$ ) in allE)
apply (erule impE)
apply assumption
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct2)
apply (drule conjunct1)
apply (erule-tac  $x = p$  in allE)
apply (erule-tac  $x = SOME \ k.$ 
   $k \notin (\bigcup x \in \text{extend } S \ C \ f \ n \cup \{Exists \ p\}. \text{params } x)$  in allE)
apply (erule impE)
apply (subgoal-tac  $\neg$  finite ( $-\ (\bigcup x \in \text{extend } S \ C \ f \ n \cup \{Exists \ p\}. \text{params } x)$ ))
prefer 2
apply simp
apply (rule ballI)
apply (drule-tac  $A = - \ ?S$  in infinite-nonempty)
apply (erule exE)
apply (rule someI2)
apply (simp only: Compl-iff [symmetric])
apply fast
apply simp
done

```

The main theorem about *Extend* says that if C is an alternative consistency property that is of finite character, S is consistent and S uses only finitely many parameters, then *Extend* $S \ C \ f$ is again consistent.

theorem *Extend-in-C*: *alt-consistency* $C \implies$ *finite-char* $C \implies$
 $S \in C \implies \neg$ *finite* $(-\ (\bigcup p \in S. \text{params } p)) \implies$ *Extend* $S\ C\ f \in C$
apply (*unfold Extend-def*)
apply (*erule chain-union-closed*)
apply (*rule is-chain-extend*)
apply (*rule allI*)
by (*rule extend-in-C*)

theorem *Extend-subset*: $S \subseteq$ *Extend* $S\ C\ f$
apply (*rule subsetI*)
apply (*simp add: Extend-def*)
apply (*rule-tac x=0 in exI*)
apply *simp*
done

The *Extend* function yields a maximal set:

definition

maximal :: 'a set \implies 'a set set \implies bool **where**
maximal $S\ C = (\forall S' \in C. S \subseteq S' \longrightarrow S = S')$

theorem *extend-maximal*: $\forall y. \exists n. y = f\ n \implies$
finite-char $C \implies$ *maximal* $(\text{Extend } S\ C\ f)\ C$
apply (*simp add: maximal-def Extend-def*)
apply (*rule ballI impI*)
apply (*rule subset-antisym*)
apply *assumption*
apply (*rule ccontr*)
apply (*subgoal-tac* $\exists z. z \in S' \wedge z \notin (\bigcup x. \text{extend } S\ C\ f\ x)$)
prefer 2
apply *blast*
apply (*erule exE conjE*)
apply (*erule-tac x=z in allE*)
apply (*erule exE*)
apply (*subgoal-tac* $\text{extend } S\ C\ f\ n \cup \{f\ n\} \subseteq S'$)
prefer 2
apply *simp*
apply (*rule subset-trans*)
prefer 2
apply *assumption*
apply (*rule UN-upper [OF UNIV-I]*)
apply (*erule finite-char-closed*)
apply (*unfold subset-closed-def*)
apply (*erule bspec*)
apply *assumption*
apply (*erule-tac x=?a \cup ?b in allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule-tac P=?a \in ?b in notE*)
apply (*rule-tac a=Suc n in UN-I [OF UNIV-I]*)

apply simp
done

6.6 Hintikka sets and Herbrand models

A Hintikka set is defined as follows:

definition

hintikka :: ('a, 'b) form set \Rightarrow bool **where**

hintikka H =

$$\begin{aligned} & ((\forall p \ ts. \neg (Pred \ p \ ts \in H \wedge Neg \ (Pred \ p \ ts) \in H)) \wedge \\ & FF \notin H \wedge Neg \ TT \notin H \wedge \\ & (\forall Z. Neg \ (Neg \ Z) \in H \longrightarrow Z \in H) \wedge \\ & (\forall A \ B. And \ A \ B \in H \longrightarrow A \in H \wedge B \in H) \wedge \\ & (\forall A \ B. Neg \ (Or \ A \ B) \in H \longrightarrow Neg \ A \in H \wedge Neg \ B \in H) \wedge \\ & (\forall A \ B. Or \ A \ B \in H \longrightarrow A \in H \vee B \in H) \wedge \\ & (\forall A \ B. Neg \ (And \ A \ B) \in H \longrightarrow Neg \ A \in H \vee Neg \ B \in H) \wedge \\ & (\forall A \ B. Impl \ A \ B \in H \longrightarrow Neg \ A \in H \vee B \in H) \wedge \\ & (\forall A \ B. Neg \ (Impl \ A \ B) \in H \longrightarrow A \in H \wedge Neg \ B \in H) \wedge \\ & (\forall P \ t. closedt \ 0 \ t \longrightarrow Forall \ P \in H \longrightarrow subst \ P \ t \ 0 \in H) \wedge \\ & (\forall P \ t. closedt \ 0 \ t \longrightarrow Neg \ (Exists \ P) \in H \longrightarrow Neg \ (subst \ P \ t \ 0) \in H) \wedge \\ & (\forall P. Exists \ P \in H \longrightarrow (\exists t. closedt \ 0 \ t \wedge subst \ P \ t \ 0 \in H)) \wedge \\ & (\forall P. Neg \ (Forall \ P) \in H \longrightarrow (\exists t. closedt \ 0 \ t \wedge Neg \ (subst \ P \ t \ 0) \in H))) \end{aligned}$$

In Herbrand models, each *closed* term is interpreted by itself. We introduce a new datatype *hterm* (“Herbrand terms”), which is similar to the datatype *term* introduced in §2, but without variables. We also define functions for converting between closed terms and Herbrand terms.

datatype 'a hterm = HApp 'a 'a hterm list

primrec

term-of-hterm :: 'a hterm \Rightarrow 'a term

and *terms-of-hterms* :: 'a hterm list \Rightarrow 'a term list

where

term-of-hterm (HApp a hts) = App a (*terms-of-hterms* hts)

| *terms-of-hterms* [] = []

| *terms-of-hterms* (ht # hts) = *term-of-hterm* ht # *terms-of-hterms* hts

theorem *herbrand-evalt* [simp]:

closedt 0 t \Longrightarrow *term-of-hterm* (*evalt* e HApp t) = t

closedts 0 ts \Longrightarrow *terms-of-hterms* (*evalts* e HApp ts) = ts

by (*induct* t **and** ts) *simp-all*

theorem *herbrand-evalt'* [simp]:

evalt e HApp (*term-of-hterm* ht) = ht

evalts e HApp (*terms-of-hterms* hts) = hts

by (*induct* ht **and** hts) *simp-all*

theorem *closed-hterm* [simp]:

closedt 0 (term-of-hterm (ht::'a hterm))
closedts 0 (terms-of-hterms (hts::'a hterm list))
by (*induct ht and hts*) *simp-all*

theorem *measure-size-eq* [*simp*]: $((x, y) \in \text{measure } f) = (f\ x < f\ y)$
by (*simp add: measure-def inv-image-def*)

We can prove that Hintikka sets are satisfiable in Herbrand models. Note that this theorem cannot be proved by a simple structural induction (as claimed in Fitting's book), since a parameter substitution has to be applied in the cases for quantifiers. However, since parameter substitution does not change the size of formulae, the theorem can be proved by well-founded induction on the size of the formula p .

theorem *hintikka-model*: *hintikka H* \implies
 $(p \in H \longrightarrow \text{closed } 0\ p \longrightarrow$
 $\text{eval } e\ H\text{App } (\lambda a\ ts.\ \text{Pred } a\ (\text{terms-of-hterms } ts) \in H)\ p) \wedge$
 $(\text{Neg } p \in H \longrightarrow \text{closed } 0\ p \longrightarrow$
 $\text{eval } e\ H\text{App } (\lambda a\ ts.\ \text{Pred } a\ (\text{terms-of-hterms } ts) \in H)\ (\text{Neg } p))$
apply (*unfold hintikka-def*)
apply (*rule-tac r=measure size and a=p in wf-induct*)
apply (*simp (no-asm)*)
apply (*case-tac x*)
apply *hypsubst*
apply (*rule conjI*)
apply (*drule conjunct2*)
apply (*drule conjunct1*)
apply *iprover*
apply (*simp (no-asm)*)
apply *hypsubst*
apply (*simp (no-asm)*)
apply *iprover*
apply *hypsubst*
apply (*simp (no-asm)*)
apply (*drule conjunct1*)
apply *iprover*
apply *hypsubst*
apply (*rule conjI impI*)+
apply (*drule conjunct2, drule conjunct2,*
 $\text{drule conjunct2, drule conjunct2,}$
 $\text{drule conjunct1, erule allE, erule allE,}$
 $\text{erule impE, assumption})$
apply *simp*
apply (*rule impI*)+
apply (*drule conjunct2, drule conjunct2,*
 $\text{drule conjunct2, drule conjunct2,}$
 $\text{drule conjunct2, drule conjunct2,}$
 $\text{drule conjunct2, drule conjunct1,}$
 $\text{erule allE, erule allE,}$
 $\text{erule impE, assumption})$

```

apply fastsimp
apply hypsubst
apply (rule conjI impI)+
apply (drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct1, erule allE, erule allE,
  erule impE, assumption)
apply fastsimp
apply (rule impI)+
apply (drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct1, erule allE, erule allE,
  erule impE, assumption)
apply simp
apply hypsubst
apply (rule conjI impI)+
apply (drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct1, erule allE, erule allE,
  erule impE, assumption)
apply fastsimp
apply (rule impI)+
apply (drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct1,
  erule allE, erule allE,
  erule impE, assumption)
apply simp
apply (rule conjI)
apply (erule thin-rl)
apply simp
apply hypsubst
apply (rule impI)+
apply (drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct1,
  erule allE, erule impE, assumption)
apply simp
apply hypsubst
apply (simp (no-asm))
apply (rule conjI impI allI)+
apply (drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,
  drule conjunct2, drule conjunct2,

```

```

    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct1)
apply (erule-tac x=subst form (term-of-hterm z) 0 in allE)
apply simp
apply (rule impI)+
apply (drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2)
apply (erule allE, erule impE, assumption, erule exE)
apply (erule-tac x=subst form t 0 in allE)
apply fastsimp
apply hypsubst
apply (simp (no-asm))
apply (rule conjI impI allI)+
apply (drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct1)
apply (erule allE, erule impE, assumption, erule exE)
apply (erule-tac x=subst form t 0 in allE)
apply fastsimp
apply (rule impI allI)+
apply (drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct2,
    drule conjunct2, drule conjunct1)
apply (erule-tac x=subst form (term-of-hterm z) 0 in allE)
apply simp
done

```

Using the maximality of *Extend S C f*, we can show that *Extend S C f* yields Hintikka sets:

theorem *extend-hintikka*:

```

assumes fin-ch: finite-char C
and infin-p:  $\neg$  finite ( $\bigcup_{p \in S} \text{params } p$ )
and surj:  $\forall y. \exists n. y = f n$ 
shows alt-consistency C  $\implies$  S  $\in$  C  $\implies$  hintikka (Extend S C f)
apply (insert extend-maximal [OF surj fin-ch, of S])
apply (frule Extend-in-C)

```

apply (*rule fin-ch*)
apply *assumption*
apply (*rule infin-p*)
apply (*unfold alt-consistency-def maximal-def hintikka-def*)
apply (*erule-tac x=Extend S C f in allE,*
erule impE, assumption)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE', fast*)
apply (*erule conjE'*)
apply (*rule allI impI*)
apply (*erule-tac x=P in allE*)
apply (*simp only: Extend-def*)
apply (*insert surj*)
apply (*erule-tac x=Exists P in allE*)
apply (*erule exE*)
apply (*subgoal-tac extend S C f n \cup {f n} \in C*)
prefer 2
apply (*cut-tac finite-char-closed [OF fin-ch]*)
apply (*unfold subset-closed-def*)
apply (*drule-tac x= \bigcup n. extend S C f n in bspec,*
assumption, erule allE, erule mp)
apply *simp*
apply *fast*
apply (*rule-tac*
x=(App (SOME k. k \notin (\bigcup p \in extend S C f n \cup {f n}. params p)) []) in exI)
apply (*rule conjI*)
apply *simp*
apply (*rule-tac a=Suc n in UN-I [OF UNIV-I]*)
apply *simp*
apply (*rule conjI impI*)
apply (*erule exE*)
apply *simp*
apply (*rule impI*)
apply (*erule-tac x=P in allE*)
apply *simp*
apply (*drule sym*)
apply *simp*
apply *fast*
apply (*rule allI impI*)

```

apply (erule-tac x=P in allE)
apply (simp only: Extend-def)
apply (insert surj)
apply (erule-tac x=Neg (Forall P) in allE)
apply (erule exE)
apply (subgoal-tac extend S C f n  $\cup$  {f n}  $\in$  C)
prefer 2
apply (cut-tac finite-char-closed [OF fin-ch])
apply (unfold subset-closed-def)
apply (drule-tac x= $\bigcup$  n. extend S C f n in bspec,
  assumption, erule allE, erule mp)
apply simp
apply fast
apply (rule-tac
  x=(App (SOME k. k  $\notin$  ( $\bigcup$  p  $\in$  extend S C f n  $\cup$  {f n}. params p)) [])) in exI)
apply (rule conjI)
apply simp
apply (rule-tac a=Suc n in UN-I [OF UNIV-I])
apply simp
apply (rule conjI impI)+
apply (erule exE)+
apply simp
apply (drule sym)
apply simp
apply (rule impI)
apply (erule-tac P= $\%x$ . ?l  $\neq$  ?f x and x=P in allE)
apply simp
done

```

6.7 Model existence theorem

Since the result of extending S is a superset of S , it follows that each consistent set S has a Herbrand model:

theorem *model-existence*:

```

consistency C  $\implies$  S  $\in$  C  $\implies$   $\neg$  finite ( $\neg$  ( $\bigcup$  p  $\in$  S. params p))  $\implies$ 
  p  $\in$  S  $\implies$  closed 0 p  $\implies$  eval e HApp ( $\lambda$ a ts.
    Pred a (terms-of-hterms ts)  $\in$  Extend S
      (mk-finite-char (mk-alt-consistency (close C))) diag-form') p
apply (rule hintikka-model [THEN conjunct1, THEN mp, THEN mp])
apply (rule extend-hintikka)
apply (rule finite-char)
apply assumption
apply (rule allI)
apply (rule-tac x=undiaform' y in exI)
apply simp
apply (rule finite-alt-consistency)
apply (rule alt-consistency)
apply (erule close-consistency)
apply (rule mk-alt-consistency-closed)

```

```

apply (rule close-closed)
apply (drule close-subset [THEN subsetD])
apply (drule mk-alt-consistency-subset [THEN subsetD])
apply (erule finite-char-subset
  [OF mk-alt-consistency-closed, OF close-closed, THEN subsetD])
apply (erule Extend-subset [THEN subsetD])
apply assumption
done

```

6.8 Completeness for Natural Deduction

Thanks to the model existence theorem, we can now show the completeness of the natural deduction calculus introduced in §4. In order for the model existence theorem to be applicable, we have to prove that the set of sets that are consistent with respect to \vdash is a consistency property:

```

theorem deriv-consistency:
  assumes inf-param:  $\neg$  finite (UNIV::'a set)
  shows consistency {S::('a, 'b) form set.  $\exists G. S = \text{set } G \wedge \neg G \vdash FF$ }
  apply (unfold consistency-def)
  apply (rule allI impI)+
  apply simp
  apply (erule exE)
  apply (erule conjE)
  apply simp
  apply (rule conjI allI impI notI)+
  apply (erule notE)
  apply (rule FFE)
  apply (rule-tac a=Pred p ts in NegE)
  apply (rule Assum)
  apply (simp add: mem-iff)
  apply (rule Assum)
  apply (simp add: mem-iff)
  apply (rule conjI notI)+
  apply (erule notE)
  apply (rule FFE)
  apply (rule Assum)
  apply (simp add: mem-iff)
  apply (rule conjI notI)+
  apply (erule notE)
  apply (rule FFE)
  apply (rule-tac a=TT in NegE)
  apply (rule Assum)
  apply (simp add: mem-iff)
  apply (rule TT)
  apply (rule conjI allI impI)+
  apply (rule-tac x=Z # G in exI)
  apply simp
  apply (rule notI)

```

```

apply (erule notE)
apply (erule cut')
apply (rule Class)
apply (rule-tac a=Neg Z in NegE)
apply (rule Assum)
apply (simp add: mem-iff)
apply (rule Assum)
apply simp
apply (rule allI impI conjI)+
apply (rule-tac x=A # B # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (erule cut' [OF cut'])
apply (rule-tac b=B in AndE1)
apply (rule Assum)
apply (simp add: mem-iff)
apply (rule-tac a=A in AndE2)
apply (rule Assum)
apply (simp add: mem-iff)
apply (rule allI impI conjI)+
apply (rule-tac x=Neg A # Neg B # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (erule cut' [OF cut'])
apply (rule NegI)
apply (rule-tac a=Or A B in NegE)
apply (rule Assum)
apply (simp add: mem-iff)
apply (rule OrI1)
apply (rule Assum)
apply simp
apply (rule NegI)
apply (rule-tac a=Or A B in NegE)
apply (rule Assum)
apply (simp add: mem-iff)
apply (rule OrI2)
apply (rule Assum)
apply simp
apply (rule allI impI conjI)+
apply (rule ccontr)
apply simp
apply (erule conjE notE)+
apply (rule-tac a=A and b=B in OrE)
apply (rule Assum)
apply (simp add: mem-iff)
apply simp
apply simp

```

apply (*rule allI impI conjI*)
apply (*rule ccontr*)
apply *simp*
apply (*erule conjE notE*)
apply (*subgoal-tac G ⊢ Or (Neg A) (Neg B)*)
apply (*erule-tac a=Neg A and b=Neg B in OrE*)
apply *simp*
apply *simp*
apply (*rule Class[⋀]*)
apply (*rule OrI1*)
apply (*rule NegI*)
apply (*rule-tac a=Or (Neg A) (Neg B) in NegE*)
apply (*rule Assum, simp*)
apply (*rule OrI2*)
apply (*rule NegI*)
apply (*rule-tac a=And A B in NegE*)
apply (*rule Assum, simp add: mem-iff*)
apply (*rule AndI*)
apply (*rule Assum, simp*)
apply (*rule Assum, simp*)
apply (*rule allI impI conjI*)
apply (*rule ccontr*)
apply *simp*
apply (*erule conjE notE*)
apply (*subgoal-tac G ⊢ Or (Neg A) B*)
apply (*erule-tac a=Neg A and b=B in OrE*)
apply *simp*
apply *simp*
apply (*rule Class[⋀]*)
apply (*rule OrI1*)
apply (*rule NegI*)
apply (*rule-tac a=Or (Neg A) B in NegE*)
apply (*rule Assum, simp*)
apply (*rule OrI2*)
apply (*rule-tac a=A in ImplE*)
apply (*rule Assum, simp add: mem-iff*)
apply (*rule Assum, simp*)
apply (*rule allI impI conjI*)
apply (*rule-tac x=A # Neg B # G in exI*)
apply *simp*
apply (*rule notI*)
apply (*erule notE*)
apply (*erule cut' [OF cut[⋀]]*)
apply (*rule Class*)
apply (*rule-tac a=Impl A B in NegE*)
apply (*rule Assum, simp add: mem-iff*)
apply (*rule ImplI*)
apply (*rule FFE*)
apply (*rule-tac a=A in NegE*)

```

apply (rule Assum, simp)
apply (rule Assum, simp)
apply (rule NegI)
apply (rule-tac a=Impl A B in NegE)
apply (rule Assum, simp add: mem-iff)
apply (rule ImplI)
apply (rule Assum, simp)
apply (rule allI impI conjI)+
apply (rule-tac x=subst P t 0 # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (erule cut')
apply (rule ForallE)
apply (rule Assum, simp add: mem-iff)
apply (rule allI impI conjI)+
apply (rule-tac x=Neg (subst P t 0) # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (erule cut')
apply (rule NegI)
apply (rule-tac a=Exists P in NegE)
apply (rule Assum, simp add: mem-iff)
apply (rule-tac t=t in ExistsI)
apply (rule Assum, simp)
apply (rule allI impI conjI)+
apply (subgoal-tac  $\exists x. x \in - ((\bigcup p \in \text{set } G. \text{params } p) \cup \text{params } P)$ )
apply simp
apply (erule exE)
apply (rule-tac x=x in exI)
apply (rule-tac x=subst P (App x []) 0 # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (rule-tac a=P in ExistsE)
apply (rule Assum, simp add: mem-iff)
apply assumption
apply (simp add: list-all-iff)
apply simp
apply simp
apply (rule infinite-nonempty)
apply (simp only: Compl-UNIV-eq)
apply (rule Diff-infinite-finite)
apply simp
apply (rule inf-param)
apply (rule allI impI)+
apply (subgoal-tac  $\exists x. x \in - ((\bigcup p \in \text{set } G. \text{params } p) \cup \text{params } P)$ )
apply simp

```

```

apply (erule exE)
apply (rule-tac x=x in exI)
apply (rule-tac x=Neg (subst P (App x [])) 0) # G in exI)
apply simp
apply (rule notI)
apply (erule notE)
apply (rule-tac a=Neg P and n=x in ExistsE)
apply (rule Class)
apply (rule-tac a=Forall P in NegE)
apply (rule Assum, simp add: mem-iff)
apply (rule-tac n=x in ForallI)
apply (rule Class)
apply (rule-tac a=Exists (Neg P) in NegE)
apply (rule Assum, simp)
apply (rule-tac t=App x [] in ExistsI)
apply (rule Assum, simp)
apply (simp add: list-all-iff)
apply simp
apply simp
apply (simp add: list-all-iff)
apply simp
apply simp
apply (rule infinite-nonempty)
apply (simp only: Compl-UNIV-eq)
apply (rule Diff-infinite-finite)
apply simp
apply (rule inf-param)
done

```

Hence, by contradiction, we have completeness of natural deduction:

```

theorem natded-complete: closed 0 p  $\implies$  list-all (closed 0) ps  $\implies$ 
   $\forall e (f::nat \Rightarrow nat \text{ hterm list} \Rightarrow nat \text{ hterm}) (g::nat \Rightarrow nat \text{ hterm list} \Rightarrow bool).$ 
  e,f,g,ps  $\models p \implies ps \vdash p$ 
apply (rule Class)
apply (rule ccontr)
apply (subgoal-tac  $\exists e f g.$  list-all (eval e f g) (Neg p # ps))
apply (simp add: model-def)
apply iprover
apply (subgoal-tac list-all (closed 0) (Neg p # ps))
apply (simp only: list-all-iff)
apply (rule-tac x=arbitrary in exI)
apply (rule exI ballI)+
apply (rule-tac S=set (Neg p # ps) in model-existence)
apply (rule deriv-consistency)
apply (rule nat-infinite)
apply (simp del: set.simps)
apply (rule exI)
apply (rule conjI)
apply (rule refl)

```

```

apply assumption
apply (simp only: Compl-UNIV-eq)
apply (rule Diff-infinite-finite)
apply (rule finite-UN-I)
apply simp
apply simp
apply (rule nat-infinite)
apply simp
apply fast
apply simp
done

```

7 Löwenheim-Skolem theorem

Another application of the model existence theorem presented in §6.7 is the Löwenheim-Skolem theorem. It says that a set of formulae that is satisfiable in an *arbitrary model* is also satisfiable in a *Herbrand model*. The main idea behind the proof is to show that satisfiable sets are consistent, hence they must be satisfiable in a Herbrand model.

```

theorem sat-consistency: consistency { $S. \neg \text{finite } (- (\bigcup p \in S. \text{params } p)) \wedge$ 
  ( $\exists f. \forall (p::('a, 'b)\text{form}) \in S. \text{eval } e \ f \ g \ p$ )}}
apply (unfold consistency-def)
apply (rule allI impI)+
apply (erule CollectE conjE)+
apply (rule conjI)
apply (rule allI notI)+
apply (erule conjE exE)+
apply (frule-tac x=Pred p ts in bspec)
apply assumption
apply (frule-tac x=Neg (Pred p ts) in bspec)
apply assumption
apply simp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule allI impI)+
apply (erule exE)+
apply (frule bspec)
apply assumption
apply simp

```

```

apply (erule disjE)
apply (rule disjI1)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule disjI2)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule conjI)
apply (rule allI impI)+
apply (erule exE)+
apply (frule bspec)
apply assumption
apply (simp del: disj-not1)
apply (erule disjE)
apply (rule disjI1)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule disjI2)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule conjI)
apply (rule allI impI)+
apply (erule exE)+
apply (frule bspec)
apply assumption
apply simp
apply (drule iffD1 [OF imp-conv-disj])
apply (erule disjE)
apply (rule disjI1)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule disjI2)
apply (rule exI)+
apply (rule conjI)
apply assumption
apply assumption
apply (rule conjI)
apply fastsimp
apply (rule conjI)

```

```

apply fastsimp
apply (rule conjI)
apply fastsimp
apply (rule conjI)
apply (rule allI impI)+
apply (erule exE)+
apply (frule bspec)
apply assumption
apply (frule-tac A=- ?S in infinite-nonempty)
apply simp
apply (erule exE)+
apply (rule-tac x=x in exI)
apply (rule-tac x=f(x:=λy. z) in exI)
apply (frule-tac P=λy. x ∉ params y in bspec)
apply assumption
apply simp
apply (rule allI impI)+
apply (erule exE)+
apply (frule bspec)
apply assumption
apply (frule-tac A=- ?S in infinite-nonempty)
apply simp
apply (erule exE)+
apply (rule-tac x=x in exI)
apply (rule-tac x=f(x:=λy. z) in exI)
apply (frule-tac P=λy. x ∉ params y in bspec)
apply assumption
apply simp
done

```

theorem *doublep-evalt* [*simp*]:

```

evalt e f (psubstt (λn::nat. 2 * n) t) = evalt e (λn. f (2*n)) t
evalts e f (psubstts (λn::nat. 2 * n) ts) = evalts e (λn. f (2*n)) ts
by (induct t and ts simp-all)

```

theorem *doublep-eval*: $\bigwedge e. \text{eval } e \text{ f } g \text{ (psubst } (\lambda n::\text{nat. } 2 * n) \text{ p) =}$

```

eval e (λn. f (2*n)) g p
by (induct p simp-all)

```

theorem *doublep-infinite-params*:

```

¬ finite (¬ (⋃ p ∈ psubst (λn::nat. 2 * n) ‘ S. params p))

```

proof (*rule infinite-super*)

```

show infinite (range (λn::nat. 2 * n + 1))

```

```

by (auto intro!: range-inj-infinite inj-onI)

```

next

```

have  $\bigwedge m \ n. \text{Suc } (2 * m) \neq 2 * n$  by arith

```

```

then show range (λn::nat. (2::nat) * n + (1::nat))

```

```

⊆ ¬ (⋃ p::(nat, 'a) form ∈ psubst (op * (2::nat)) ‘ S. params p)

```

```

by auto

```

qed

When applying the model existence theorem, there is a technical complication. We must make sure that there are infinitely many unused parameters. In order to achieve this, we encode parameters as natural numbers and multiply each parameter occurring in the set S by 2.

theorem *loewenheim-skolem*: $\forall p \in S. \text{eval } e \text{ f g } p \implies$
 $\forall p \in S. \text{closed } 0 \text{ p} \longrightarrow \text{eval } e' (\lambda n. \text{HApp } (2 * n)) (\lambda a \text{ ts.}$
 $\text{Pred } a (\text{terms-of-hterms } \text{ts}) \in \text{Extend } (\text{psubst } (\lambda n. 2 * n) ' S)$
 $(\text{mk-finite-char } (\text{mk-alt-consistency } (\text{close}$
 $\{S. \neg \text{finite } (\neg (\bigcup p \in S. \text{params } p)) \wedge$
 $(\exists f. \forall p \in S. \text{eval } e \text{ f g } p\}))) \text{diag-form}') p$
apply (*simp only: doublep-eval [symmetric]*)
apply (*rule ballI impI*)
apply (*rule model-existence*)
apply (*rule sat-consistency*)
apply (*rule CollectI*)
apply (*rule conjI*)
apply (*rule doublep-infinite-params*)
apply (*rule-tac x= $\lambda n. f (n \text{ div } 2)$ in exI*)
apply (*rule ballI*)
apply (*erule imageE*)
apply (*drule-tac x= xa in bspec*)
apply *assumption*
apply (*simp add: doublep-eval*)
apply (*rule doublep-infinite-params*)
apply *simp*
apply *simp*
done

References

- [1] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, second edition, 1996.