Meta-theory of first-order predicate logic

Stefan Berghofer

December 12, 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 4

3 Semantics **6**

4 Proof calculus **8**

5 Correctness **10**

6 Completeness **10**
- 6.1 Consistent sets 11
- 6.2 Closure under subsets 12
- 6.3 Finite character 13
- 6.4 Enumerating datatypes 13
  - 6.4.1 Enumerating pairs of natural numbers 13
  - 6.4.2 Enumerating trees 15
  - 6.4.3 Enumerating lists 15
  - 6.4.4 Enumerating terms 16
- 6.5 Extension to maximal consistent sets 18
- 6.6 Hintikka sets and Herbrand models 20

6.7 Model existence theorem 21
6.8 Completeness for Natural Deduction 22

7 Löwenheim-Skolem theorem 22

1 Miscellaneous Utilities

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

theorem *conjE'*: $P \wedge Q \Longrightarrow (P \Longrightarrow P') \Longrightarrow (Q \Longrightarrow Q') \Longrightarrow P' \wedge Q'$
⟨*proof*⟩

theorem *conjE''*: $(\forall x.\ P\ x \longrightarrow Q\ x \wedge R\ x) \Longrightarrow$
$((\forall x.\ P\ x \longrightarrow Q\ x) \Longrightarrow Q') \Longrightarrow ((\forall x.\ P\ x \longrightarrow R\ x) \Longrightarrow R') \Longrightarrow Q' \wedge R'$
⟨*proof*⟩

Some facts about (in)finite sets

theorem [*simp*]: $-\ A \cap B = B - A$ ⟨*proof*⟩

theorem *Compl-UNIV-eq*: $-\ A = \mathit{UNIV} - A$ ⟨*proof*⟩

theorem *infinite-nonempty*: $\neg\ \mathit{finite}\ A \Longrightarrow \exists x.\ x \in A$
⟨*proof*⟩

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* ⇒ *'a term* ⇒ *bool*
  and *closedts* :: *nat* ⇒ *'a term list* ⇒ *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* ∧ *closedts m ts*)

primrec
  closed :: *nat* ⇒ (*'a*, *'b*) *form* ⇒ *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* ∧ *closed m q*)
| *closed m* (*Or p q*) = (*closed m p* ∧ *closed m q*)
| *closed m* (*Impl p q*) = (*closed m p* ∧ *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* ≤ *j*
  shows *closedt i* (*t*::*'a term*) ⟹ *closedt j t*
  and *closedts i* (*ts*::*'a term list*) ⟹ *closedts j ts* ⟨*proof*⟩

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* ⇒ *'a term* ⇒ *nat* ⇒ *'a term* (-[-'/-] [*300*, *0*, *0*] *300*)
  and *substts* :: *'a term list* ⇒ *'a term* ⇒ *nat* ⇒ *'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* ⇒ *'a term*
  and *liftts* :: *'a term list* ⇒ *'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* ⇒ *'a term* ⇒ *nat* ⇒ (*'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*) ⟹ *closedt 0* (*liftt t*)
  closedts 0 (*ts*::*'a term list*) ⟹ *closedts 0* (*liftts ts*)
  ⟨*proof*⟩

theorem *subst-closedt* [*simp*]: **assumes** *u*: *closedt 0 u*
  shows *closedt* (*Suc i*) *t* ⟹ *closedt i* (*t*[*u*/*i*])
  and *closedts* (*Suc i*) *ts* ⟹ *closedts i* (*ts*[*u*/*i*]) ⟨*proof*⟩

theorem *subst-closed* [*simp*]:
  closedt 0 t ⟹ *closed* (*Suc i*) *p* ⟹ *closed i* (*p*[*t*/*i*])
  ⟨*proof*⟩

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

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 term ⇒ 'a set
  and paramsts :: 'a term list ⇒ 'a set
where
  paramst (Var n) = {}
| paramst (App a ts) = {a} ∪ paramsts ts
| paramsts [] = {}
| paramsts (t # ts) = (paramst t ∪ paramsts ts)

primrec
  params :: ('a, 'b) form ⇒ 'a set
where
  params FF = {}
| params TT = {}
| params (Pred b ts) = paramsts ts
| params (And p q) = params p ∪ params q
| params (Or p q) = params p ∪ params q
| params (Impl p q) = params p ∪ 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
  psubstt :: ('a ⇒ 'c) ⇒ 'a term ⇒ 'c term
  and psubstts :: ('a ⇒ 'c) ⇒ 'a term list ⇒ 'c term list
where
  psubstt f (Var i) = Var i
| psubstt f (App x ts) = App (f x) (psubstts f ts)
| psubstts f [] = []
| psubstts f (t # ts) = psubstt f t # psubstts f ts

primrec
  psubst :: ('a ⇒ 'c) ⇒ ('a, 'b) form ⇒ ('c, 'b) form
where
  psubst f FF = FF
| psubst f TT = TT
| psubst f (Pred b ts) = Pred b (psubstts f ts)
| psubst f (And p q) = And (psubst f p) (psubst f q)
| psubst f (Or p q) = Or (psubst f p) (psubst f q)
| psubst f (Impl p q) = Impl (psubst f p) (psubst f q)
| psubst f (Neg p) = Neg (psubst f p)
| psubst f (Forall p) = Forall (psubst f p)
| psubst f (Exists p) = Exists (psubst f p)

theorem psubstt-closed [simp]:
  closedt i (psubstt f t) = closedt i t
```

closedts i (*psubstts f ts*) = *closedts i ts*
⟨*proof*⟩

theorem *psubst-closed* [*simp*]:
closed i (*psubst f p*) = *closed i p*
⟨*proof*⟩

theorem *psubstt-subst* [*simp*]:
psubstt f (*substt t u i*) = *substt* (*psubstt f t*) (*psubstt f u*) *i*
psubstts f (*substts ts u i*) = *substts* (*psubstts f ts*) (*psubstt f u*) *i*
⟨*proof*⟩

theorem *psubstt-lift* [*simp*]:
psubstt f (*liftt t*) = *liftt* (*psubstt f t*)
psubstts f (*liftts ts*) = *liftts* (*psubstts f ts*)
⟨*proof*⟩

theorem *psubst-subst* [*simp*]:
psubst f (*subst P t i*) = *subst* (*psubst f P*) (*psubstt f t*) *i*
⟨*proof*⟩

theorem *psubstt-upd* [*simp*]:
$x \notin$ *paramst* (*t*::'*a term*) $\Longrightarrow$ *psubstt* ($f(x{:=}y)$) *t* = *psubstt f t*
$x \notin$ *paramsts* (*ts*::'*a term list*) $\Longrightarrow$ *psubstts* ($f(x{:=}y)$) *ts* = *psubstts f ts*
⟨*proof*⟩

theorem *psubst-upd* [*simp*]: $x \notin$ *params P* $\Longrightarrow$ *psubst* ($f(x{:=}y)$) *P* = *psubst f P*
⟨*proof*⟩

theorem *psubstt-id* [*simp*]: *psubstt* (%*x*. *x*) (*t*::'*a term*) = *t*
psubstts (%*x*. *x*) (*ts*::'*a term list*) = *ts*
⟨*proof*⟩

theorem *psubst-id* [*simp*]: *psubst* (%*x*. *x*) = (%*p*. *p*)
⟨*proof*⟩

theorem *psubstt-image* [*simp*]:
paramst (*psubstt f t*) = *f* ‘ *paramst t*
paramsts (*psubstts f ts*) = *f* ‘ *paramsts ts*
⟨*proof*⟩

theorem *psubst-image* [*simp*]: *params* (*psubst f p*) = *f* ‘ *params p*
⟨*proof*⟩

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

lemma *shift-eq* [*simp*]: $i = j \Longrightarrow (e\langle i{:}T\rangle)\ j = T$
⟨*proof*⟩

lemma *shift-gt* [*simp*]: $j < i \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ j$
⟨*proof*⟩

lemma *shift-lt* [*simp*]: $i < j \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ (j - 1)$
⟨*proof*⟩

lemma *shift-commute* [*simp*]: $e\langle i{:}U\rangle\langle 0{:}T\rangle = e\langle 0{:}T\rangle\langle Suc\ i{:}U\rangle$
⟨*proof*⟩

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
$model :: (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c\ list \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c\ list \Rightarrow bool) \Rightarrow$

$('a,\ 'b)\ \mathit{form\ list} \Rightarrow ('a,\ 'b)\ \mathit{form} \Rightarrow \mathit{bool}\ \ (\text{-,-,-,-} \models \text{-}\ [50,50]\ 50)$ **where**
$(e,f,g,ps \models p) = (\mathit{list\text{-}all}\ (\mathit{eval}\ e\ f\ g)\ ps \longrightarrow \mathit{eval}\ e\ f\ g\ p)$

The following substitution lemmas relate substitution and evaluation functions:

theorem *subst-lemma'* [*simp*]:
$\mathit{evalt}\ e\ f\ (\mathit{substt}\ t\ u\ i) = \mathit{evalt}\ (e\langle i{:}\mathit{evalt}\ e\ f\ u\rangle)\ f\ t$
$\mathit{evalts}\ e\ f\ (\mathit{substts}\ ts\ u\ i) = \mathit{evalts}\ (e\langle i{:}\mathit{evalt}\ e\ f\ u\rangle)\ f\ ts$
⟨*proof*⟩

theorem *lift-lemma* [*simp*]:
$\mathit{evalt}\ (e\langle 0{:}z\rangle)\ f\ (\mathit{liftt}\ t) = \mathit{evalt}\ e\ f\ t$
$\mathit{evalts}\ (e\langle 0{:}z\rangle)\ f\ (\mathit{liftts}\ ts) = \mathit{evalts}\ e\ f\ ts$
⟨*proof*⟩

theorem *subst-lemma* [*simp*]:
$\bigwedge e\ i\ t.\ \mathit{eval}\ e\ f\ g\ (\mathit{subst}\ a\ t\ i) = \mathit{eval}\ (e\langle i{:}\mathit{evalt}\ e\ f\ t\rangle)\ f\ g\ a$
⟨*proof*⟩

theorem *upd-lemma'* [*simp*]:
$n \notin \mathit{paramst}\ t \Longrightarrow \mathit{evalt}\ e\ (f(n{:=}x))\ t = \mathit{evalt}\ e\ f\ t$
$n \notin \mathit{paramsts}\ ts \Longrightarrow \mathit{evalts}\ e\ (f(n{:=}x))\ ts = \mathit{evalts}\ e\ f\ ts$
⟨*proof*⟩

theorem *upd-lemma* [*simp*]:
$n \notin \mathit{params}\ p \Longrightarrow \mathit{eval}\ e\ (f(n{:=}x))\ g\ p = \mathit{eval}\ e\ f\ g\ p$
⟨*proof*⟩

theorem *list-upd-lemma* [*simp*]: $\mathit{list\text{-}all}\ (\lambda p.\ n \notin \mathit{params}\ p)\ G \Longrightarrow$
$\mathit{list\text{-}all}\ (\mathit{eval}\ e\ (f(n{:=}x))\ g)\ G = \mathit{list\text{-}all}\ (\mathit{eval}\ e\ f\ g)\ G$
⟨*proof*⟩

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

theorem *ex-all-commute-eval*:
$\mathit{eval}\ e\ f\ g\ (\mathit{Impl}\ (\mathit{Exists}\ (\mathit{Forall}\ (\mathit{Pred}\ p\ [\mathit{Var}\ 1,\ \mathit{Var}\ 0])))$
$(\mathit{Forall}\ (\mathit{Exists}\ (\mathit{Pred}\ p\ [\mathit{Var}\ 0,\ \mathit{Var}\ 1]))))$
⟨*proof*⟩

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
$\mathit{deriv} :: ('a,\ 'b)\ \mathit{form\ list} \Rightarrow ('a,\ 'b)\ \mathit{form} \Rightarrow \mathit{bool}\ (\text{-} \vdash \text{-}\ [50,50]\ 50)$
where
$\mathit{Assum}{:}\ a\ \mathit{mem}\ G \Longrightarrow G \vdash a$

| TTI: $G \vdash TT$
| FFE: $G \vdash FF \Longrightarrow G \vdash a$
| $NegI$: $a \# G \vdash FF \Longrightarrow G \vdash Neg\ a$
| $NegE$: $G \vdash Neg\ a \Longrightarrow G \vdash a \Longrightarrow G \vdash FF$
| $Class$: $Neg\ a \# G \vdash FF \Longrightarrow G \vdash a$
| $AndI$: $G \vdash a \Longrightarrow G \vdash b \Longrightarrow G \vdash And\ a\ b$
| $AndE1$: $G \vdash And\ a\ b \Longrightarrow G \vdash a$
| $AndE2$: $G \vdash And\ a\ b \Longrightarrow G \vdash b$
| $OrI1$: $G \vdash a \Longrightarrow G \vdash Or\ a\ b$
| $OrI2$: $G \vdash b \Longrightarrow G \vdash Or\ a\ b$
| OrE: $G \vdash Or\ a\ b \Longrightarrow a \# G \vdash c \Longrightarrow b \# G \vdash c \Longrightarrow G \vdash c$
| $ImplI$: $a \# G \vdash b \Longrightarrow G \vdash Impl\ a\ b$
| $ImplE$: $G \vdash Impl\ a\ b \Longrightarrow G \vdash a \Longrightarrow G \vdash b$
| $ForallI$: $G \vdash a[App\ n\ []/0] \Longrightarrow list\text{-}all\ (\lambda p.\ n \notin params\ p)\ G \Longrightarrow$
$n \notin params\ a \Longrightarrow G \vdash Forall\ a$
| $ForallE$: $G \vdash Forall\ a \Longrightarrow G \vdash a[t/0]$
| $ExistsI$: $G \vdash a[t/0] \Longrightarrow G \vdash Exists\ a$
| $ExistsE$: $G \vdash Exists\ a \Longrightarrow a[App\ n\ []/0] \# G \vdash b \Longrightarrow$
$list\text{-}all\ (\lambda p.\ n \notin params\ p)\ G \Longrightarrow n \notin params\ a \Longrightarrow n \notin params\ b \Longrightarrow G \vdash b$

The following derived inference rules are sometimes useful in applications.

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

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

theorem $Class'$: $Neg\ A \# G \vdash A \Longrightarrow G \vdash A$
⟨*proof*⟩

theorem $ForallE'$: $G \vdash Forall\ a \Longrightarrow subst\ a\ t\ 0 \# G \vdash B \Longrightarrow G \vdash B$
⟨*proof*⟩

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: $[] \vdash Or\ (Pred\ p\ [])\ (Neg\ (Pred\ p\ []))$
⟨*proof*⟩

theorem *ex-all-commute*:
$([]::(nat, 'b)\ form\ list) \vdash Impl\ (Exists\ (Forall\ (Pred\ p\ [Var\ 1,\ Var\ 0])))$
$(Forall\ (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1])))$
⟨*proof*⟩

theorem $drinker$: $([]::(nat, 'b)\ form\ list) \vdash$
$Exists\ (Impl\ (Pred\ P\ [Var\ 0])\ (Forall\ (Pred\ P\ [Var\ 0])))$
⟨*proof*⟩

theorem *peirce*:

$[] \vdash \mathit{Impl}\ (\mathit{Impl}\ (\mathit{Impl}\ (\mathit{Pred}\ P\ [])\ (\mathit{Pred}\ Q\ []))\ (\mathit{Pred}\ P\ []))\ (\mathit{Pred}\ P\ [])$
⟨*proof*⟩

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*: $G \vdash p \Longrightarrow \forall\, e\ f\ g.\ e,f,g,G \models p$
⟨*proof*⟩

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. $ps \models p \Longrightarrow ps \vdash p$
2. All consistent sets are satisfiable

The latter property is called the *model existence theorem*. To see why 2 implies 1, observe that $\mathit{Neg}\ p,\ ps \nvdash \mathit{FF}$ implies that $\mathit{Neg}\ p,\ ps$ is consistent, which, by the model existence theorem, implies that $\mathit{Neg}\ p,\ ps$ has a model, which in turn implies that $ps \not\models p$. By contraposition, it therefore follows from $ps \models p$ that $\mathit{Neg}\ p,\ ps \vdash \mathit{FF}$, 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. $S \nvdash \mathit{FF}$. 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 ϕ_0, ϕ_1, ϕ_2, ... 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

```
consistency :: ('a, 'b) form set set ⇒ bool where
consistency C = (∀ S. S ∈ C ⟶
  (∀ p ts. ¬ (Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S)) ∧
  FF ∉ S ∧ Neg TT ∉ S ∧
  (∀ Z. Neg (Neg Z) ∈ S ⟶ S ∪ {Z} ∈ C) ∧
  (∀ A B. And A B ∈ S ⟶ S ∪ {A, B} ∈ C) ∧
  (∀ A B. Neg (Or A B) ∈ S ⟶ S ∪ {Neg A, Neg B} ∈ C) ∧
  (∀ A B. Or A B ∈ S ⟶ S ∪ {A} ∈ C ∨ S ∪ {B} ∈ C) ∧
  (∀ A B. Neg (And A B) ∈ S ⟶ S ∪ {Neg A} ∈ C ∨ S ∪ {Neg B} ∈ C) ∧
  (∀ A B. Impl A B ∈ S ⟶ S ∪ {Neg A} ∈ C ∨ S ∪ {B} ∈ C) ∧
  (∀ A B. Neg (Impl A B) ∈ S ⟶ S ∪ {A, Neg B} ∈ C) ∧
  (∀ P t. closedt 0 t ⟶ Forall P ∈ S ⟶ S ∪ {P[t/0]} ∈ C) ∧
  (∀ P t. closedt 0 t ⟶ Neg (Exists P) ∈ S ⟶ S ∪ {Neg (P[t/0])} ∈ C) ∧
  (∀ P. Exists P ∈ S ⟶ (∃ x. S ∪ {P[App x []/0]} ∈ C)) ∧
  (∀ P. Neg (Forall P) ∈ S ⟶ (∃ x. S ∪ {Neg (P[App x []/0])} ∈ C)))
```

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

```
alt-consistency :: ('a, 'b) form set set ⇒ bool where
alt-consistency C = (∀ S. S ∈ C ⟶
  (∀ p ts. ¬ (Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S)) ∧
  FF ∉ S ∧ Neg TT ∉ S ∧
  (∀ Z. Neg (Neg Z) ∈ S ⟶ S ∪ {Z} ∈ C) ∧
  (∀ A B. And A B ∈ S ⟶ S ∪ {A, B} ∈ C) ∧
  (∀ A B. Neg (Or A B) ∈ S ⟶ S ∪ {Neg A, Neg B} ∈ C) ∧
  (∀ A B. Or A B ∈ S ⟶ S ∪ {A} ∈ C ∨ S ∪ {B} ∈ C) ∧
  (∀ A B. Neg (And A B) ∈ S ⟶ S ∪ {Neg A} ∈ C ∨ S ∪ {Neg B} ∈ C) ∧
  (∀ A B. Impl A B ∈ S ⟶ S ∪ {Neg A} ∈ C ∨ S ∪ {B} ∈ C) ∧
  (∀ A B. Neg (Impl A B) ∈ S ⟶ S ∪ {A, Neg B} ∈ C) ∧
  (∀ P t. closedt 0 t ⟶ Forall P ∈ S ⟶ S ∪ {P[t/0]} ∈ C) ∧
  (∀ P t. closedt 0 t ⟶ Neg (Exists P) ∈ S ⟶ S ∪ {Neg (P[t/0])} ∈ C) ∧
  (∀ P x. (∀ a∈S. x ∉ params a) ⟶ Exists P ∈ S ⟶
    S ∪ {P[App x []/0]} ∈ C) ∧
  (∀ P x. (∀ a∈S. x ∉ params a) ⟶ Neg (Forall P) ∈ S ⟶
```

$S \cup \{Neg\ (P[App\ x\ []/0])\} \in C))$

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

theorem *alt-consistency*:
consistency $C \Longrightarrow$ *alt-consistency* (*mk-alt-consistency* C)
⟨*proof*⟩

theorem *mk-alt-consistency-subset*: $C \subseteq$ *mk-alt-consistency* C
⟨*proof*⟩

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 $C = \{S.\ \exists S' \in C.\ S \subseteq S'\}$

definition
subset-closed :: 'a *set set* ⇒ *bool* **where**
subset-closed $C = (\forall S' \in C.\ \forall S.\ S \subseteq S' \longrightarrow S \in C)$

theorem *close-consistency*: *consistency* $C \Longrightarrow$ *consistency* (*close* C)
⟨*proof*⟩

theorem *close-closed*: *subset-closed* (*close* C)
⟨*proof*⟩

theorem *close-subset*: $C \subseteq$ *close* C
⟨*proof*⟩

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

theorem *mk-alt-consistency-closed*:
subset-closed $C \Longrightarrow$ *subset-closed* (*mk-alt-consistency* C)
⟨*proof*⟩

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

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

theorem *finite-alt-consistency*:
$\mathit{alt\text{-}consistency}\ C \Longrightarrow \mathit{subset\text{-}closed}\ C \Longrightarrow \mathit{alt\text{-}consistency}\ (\mathit{mk\text{-}finite\text{-}char}\ C)$
$\langle \mathit{proof} \rangle$

theorem *finite-char*: $\mathit{finite\text{-}char}\ (\mathit{mk\text{-}finite\text{-}char}\ C)$
$\langle \mathit{proof} \rangle$

theorem *finite-char-closed*: $\mathit{finite\text{-}char}\ C \Longrightarrow \mathit{subset\text{-}closed}\ C$
$\langle \mathit{proof} \rangle$

theorem *finite-char-subset*: $\mathit{subset\text{-}closed}\ C \Longrightarrow C \subseteq \mathit{mk\text{-}finite\text{-}char}\ C$
$\langle \mathit{proof} \rangle$

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
$\mathit{diag} :: \mathit{nat} \Rightarrow (\mathit{nat} \times \mathit{nat})$
where
$\mathit{diag}\ 0 = (0,\ 0)$
$\mid \mathit{diag}\ (\mathit{Suc}\ n) =$
$(\mathit{let}\ (x,\ y) = \mathit{diag}\ n$
$\mathit{in}\ \mathit{case}\ y\ \mathit{of}$
$0 \Rightarrow (0,\ \mathit{Suc}\ x)$

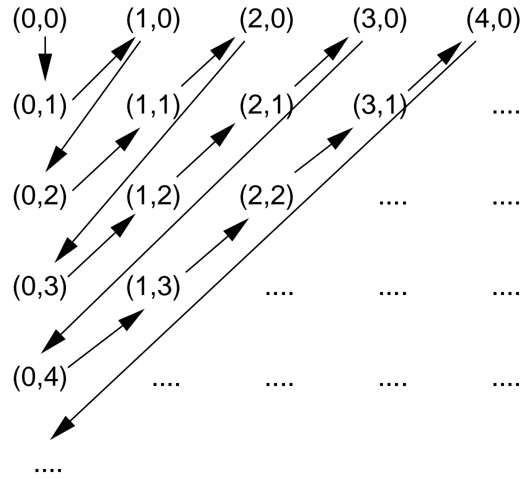

Figure 1: Cantor's method for enumerating sets of pairs

$\quad | \ \mathit{Suc}\ y \Rightarrow (\mathit{Suc}\ x,\ y))$

theorem *diag-le1*: $\mathit{fst}\ (\mathit{diag}\ (\mathit{Suc}\ n)) < \mathit{Suc}\ n$
$\langle \mathit{proof} \rangle$

theorem *diag-le2*: $\mathit{snd}\ (\mathit{diag}\ (\mathit{Suc}\ (\mathit{Suc}\ n))) < \mathit{Suc}\ (\mathit{Suc}\ n)$
$\langle \mathit{proof} \rangle$

theorem *diag-le3*: $\mathit{fst}\ (\mathit{diag}\ n) = \mathit{Suc}\ x \Longrightarrow \mathit{snd}\ (\mathit{diag}\ n) < n$
$\langle \mathit{proof} \rangle$

theorem *diag-le4*: $\mathit{fst}\ (\mathit{diag}\ n) = \mathit{Suc}\ x \Longrightarrow x < n$
$\langle \mathit{proof} \rangle$

function
$\quad \mathit{undiag} :: \mathit{nat} \times \mathit{nat} \Rightarrow \mathit{nat}$
where
$\quad \mathit{undiag}\ (0,\ 0) = 0$
$| \ \mathit{undiag}\ (0,\ \mathit{Suc}\ y) = \mathit{Suc}\ (\mathit{undiag}\ (y,\ 0))$
$| \ \mathit{undiag}\ (\mathit{Suc}\ x,\ y) = \mathit{Suc}\ (\mathit{undiag}\ (x,\ \mathit{Suc}\ y))$
$\langle \mathit{proof} \rangle$

termination
$\langle \mathit{proof} \rangle$

theorem *diag-undiag* [*simp*]: $\mathit{diag}\ (\mathit{undiag}\ (x,\ y)) = (x,\ y)$
$\langle \mathit{proof} \rangle$

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* ⇒ *btree*
where
  diag-btree n = (*case fst* (*diag n*) *of*
    0 ⇒ *Leaf* (*snd* (*diag n*))
   | *Suc x* ⇒ *Branch* (*diag-btree x*) (*diag-btree* (*snd* (*diag n*))))
  ⟨*proof*⟩

termination
  ⟨*proof*⟩

primrec
  undiag-btree :: *btree* ⇒ *nat*
where
  undiag-btree (*Leaf n*) = *undiag* (*0*, *n*)
| *undiag-btree* (*Branch t1 t2*) =
    undiag (*Suc* (*undiag-btree t1*), *undiag-btree t2*)

theorem *diag-undiag-btree* [*simp*]: *diag-btree* (*undiag-btree t*) = *t*
  ⟨*proof*⟩

declare *diag-btree.simps* [*simp del*] *undiag-btree.simps* [*simp del*]

6.4.3 Enumerating lists

fun
  list-of-btree :: (*nat* ⇒ *'a*) ⇒ *btree* ⇒ *'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-of-list :: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *btree*
where
  btree-of-list f [] = *Leaf 0*
| *btree-of-list f* (*x* # *xs*) = *Branch* (*Leaf* (*f x*)) (*btree-of-list f xs*)

definition
  diag-list :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a list* **where**
  diag-list f n = *list-of-btree f* (*diag-btree n*)

definition
undiag-list :: ('*a* ⇒ *nat*) ⇒ '*a list* ⇒ *nat* **where**
undiag-list f xs = *undiag-btree* (*btree-of-list f xs*)

theorem *diag-undiag-list* [*simp*]:
($\bigwedge x$. *d* (*u x*) = *x*) ⟹ *diag-list d* (*undiag-list u xs*) = *xs*
⟨*proof*⟩

6.4.4 Enumerating terms

fun
term-of-btree :: (*nat* ⇒ '*a*) ⇒ *btree* ⇒ '*a term*
and *term-list-of-btree* :: (*nat* ⇒ '*a*) ⇒ *btree* ⇒ '*a term list*
where
term-of-btree f (*Leaf m*) = *Var m*
| *term-of-btree f* (*Branch* (*Leaf m*) *t*) =
App (*f m*) (*term-list-of-btree f t*)
| *term-list-of-btree f* (*Leaf m*) = []
| *term-list-of-btree f* (*Branch t1 t2*) =
term-of-btree f t1 # *term-list-of-btree f t2*

primrec
btree-of-term :: ('*a* ⇒ *nat*) ⇒ '*a term* ⇒ *btree*
and *btree-of-term-list* :: ('*a* ⇒ *nat*) ⇒ '*a term list* ⇒ *btree*
where
btree-of-term f (*Var m*) = *Leaf m*
| *btree-of-term f* (*App m ts*) = *Branch* (*Leaf* (*f m*)) (*btree-of-term-list f ts*)
| *btree-of-term-list f* [] = *Leaf 0*
| *btree-of-term-list f* (*t* # *ts*) = *Branch* (*btree-of-term f t*) (*btree-of-term-list f ts*)

theorem *term-btree*: **assumes** *du*: $\bigwedge x$. *d* (*u x*) = *x*
shows *term-of-btree d* (*btree-of-term u t*) = *t*
and *term-list-of-btree d* (*btree-of-term-list u ts*) = *ts*
⟨*proof*⟩

definition
diag-term :: (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*a term* **where**
diag-term f n = *term-of-btree f* (*diag-btree n*)

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

theorem *diag-undiag-term* [*simp*]:
($\bigwedge x$. *d* (*u x*) = *x*) ⟹ *diag-term d* (*undiag-term u t*) = *t*
⟨*proof*⟩

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 (undiag-list (undiag-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-form :: (nat ⇒ 'a) ⇒ (nat ⇒ 'b) ⇒ nat ⇒ ('a, 'b) form where
  diag-form f g n = form-of-btree f g (diag-btree n)

definition
  undiag-form :: ('a ⇒ nat) ⇒ ('b ⇒ nat) ⇒ ('a, 'b) form ⇒ nat where
  undiag-form f g x = undiag-btree (btree-of-form f g x)

theorem diag-undiag-form [simp]:
```

$(\bigwedge x.\ d\ (u\ x) = x) \Longrightarrow (\bigwedge x.\ d'\ (u'\ x) = x) \Longrightarrow$
diag-form $d\ d'$ (*undiag-form* $u\ u'\ f) = f$
⟨*proof*⟩

definition
diag-form' :: *nat* ⇒ (*nat*, *nat*) *form* **where**
diag-form' = *diag-form* $(\lambda n.\ n)\ (\lambda n.\ n)$

definition
undiag-form' :: (*nat*, *nat*) *form* ⇒ *nat* **where**
undiag-form' = *undiag-form* $(\lambda n.\ n)\ (\lambda n.\ n)$

theorem *diag-undiag-form'* [*simp*]: *diag-form'* (*undiag-form'* $f) = f$
⟨*proof*⟩

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

theorem *is-chainD*: *is-chain* $f \Longrightarrow x \in f\ m \Longrightarrow x \in f\ (m + n)$
⟨*proof*⟩

theorem *is-chainD'*: *is-chain* $f \Longrightarrow x \in f\ m \Longrightarrow m \leq k \Longrightarrow x \in f\ k$
⟨*proof*⟩

theorem *chain-index*:
assumes *ch*: *is-chain* f **and** *fin*: *finite* F
shows $F \subseteq (\bigcup n.\ f\ n) \Longrightarrow \exists n.\ F \subseteq f\ n$ ⟨*proof*⟩

theorem *chain-union-closed*:
finite-char $C \Longrightarrow$ *is-chain* $f \Longrightarrow \forall n.\ f\ n \in C \Longrightarrow (\bigcup n.\ f\ n) \in C$
⟨*proof*⟩

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* ⇒ (*'a*, *'b*) *form*
where
dest-Neg (*Neg* $p) = p$

primrec
dest-Forall :: (*'a*, *'b*) *form* ⇒ (*'a*, *'b*) *form*
where

dest-Forall (*Forall p*) = *p*

primrec
dest-Exists :: (*'a*, *'b*) *form* ⇒ (*'a*, *'b*) *form*
where
dest-Exists (*Exists p*) = *p*

primrec
extend :: (*nat*, *'b*) *form set* ⇒ (*nat*, *'b*) *form set set* ⇒
(*nat* ⇒ (*nat*, *'b*) *form*) ⇒ *nat* ⇒ (*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*)
⟨*proof*⟩

theorem *finite-paramst* [*simp*]: *finite* (*paramst* (*t* :: *'a term*))
finite (*paramsts* (*ts* :: *'a term list*))
⟨*proof*⟩

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

theorem *finite-params-extend* [*simp*]:
¬ *finite* (⋂ *p* ∈ *S*. − *params p*) ⟹ ¬ *finite* (⋂ *p* ∈ *extend S C f n*. − *params p*)
⟨*proof*⟩

theorem *extend-in-C*: *alt-consistency C* ⟹
S ∈ *C* ⟹ ¬ *finite* (− (⋃ *p* ∈ *S*. *params p*)) ⟹ *extend S C f n* ∈ *C*
⟨*proof*⟩

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*: $alt\text{-}consistency\ C \Longrightarrow finite\text{-}char\ C \Longrightarrow$
$S \in C \Longrightarrow \neg\ finite\ (-\ (\bigcup p \in S.\ params\ p)) \Longrightarrow Extend\ S\ C\ f \in C$
⟨*proof*⟩

theorem *Extend-subset*: $S \subseteq Extend\ S\ C\ f$
⟨*proof*⟩

The *Extend* function yields a maximal set:

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

theorem *extend-maximal*: $\forall y.\ \exists n.\ y = f\ n \Longrightarrow$
$finite\text{-}char\ C \Longrightarrow maximal\ (Extend\ S\ C\ f)\ C$
⟨*proof*⟩

6.6 Hintikka sets and Herbrand models

A Hintikka set is defined as follows:

definition
$hintikka :: ('a, 'b)\ form\ set \Rightarrow bool$ **where**
$hintikka\ H =$
$((\forall p\ ts.\ \neg\ (Pred\ p\ ts \in H \wedge Neg\ (Pred\ p\ ts) \in H)) \wedge$
$FF \notin H \wedge Neg\ TT \notin H \wedge$
$(\forall Z.\ Neg\ (Neg\ Z) \in H \longrightarrow Z \in H) \wedge$
$(\forall A\ B.\ And\ A\ B \in H \longrightarrow A \in H \wedge B \in H) \wedge$
$(\forall A\ B.\ Neg\ (Or\ A\ B) \in H \longrightarrow Neg\ A \in H \wedge Neg\ B \in H) \wedge$
$(\forall A\ B.\ Or\ A\ B \in H \longrightarrow A \in H \vee B \in H) \wedge$
$(\forall A\ B.\ Neg\ (And\ A\ B) \in H \longrightarrow Neg\ A \in H \vee Neg\ B \in H) \wedge$
$(\forall A\ B.\ Impl\ A\ B \in H \longrightarrow Neg\ A \in H \vee B \in H) \wedge$
$(\forall A\ B.\ Neg\ (Impl\ A\ B) \in H \longrightarrow A \in H \wedge Neg\ B \in H) \wedge$
$(\forall P\ t.\ closedt\ 0\ t \longrightarrow Forall\ P \in H \longrightarrow subst\ P\ t\ 0 \in H) \wedge$
$(\forall P\ t.\ closedt\ 0\ t \longrightarrow Neg\ (Exists\ P) \in H \longrightarrow Neg\ (subst\ P\ t\ 0) \in H) \wedge$
$(\forall P.\ Exists\ P \in H \longrightarrow (\exists t.\ closedt\ 0\ t \wedge subst\ P\ t\ 0 \in H)) \wedge$
$(\forall P.\ Neg\ (Forall\ P) \in H \longrightarrow (\exists t.\ closedt\ 0\ t \wedge Neg\ (subst\ P\ t\ 0) \in H)))$

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
$term\text{-}of\text{-}hterm :: 'a\ hterm \Rightarrow 'a\ term$
and $terms\text{-}of\text{-}hterms :: 'a\ hterm\ list \Rightarrow 'a\ term\ list$
where
$term\text{-}of\text{-}hterm\ (HApp\ a\ hts) = App\ a\ (terms\text{-}of\text{-}hterms\ hts)$
$|\ terms\text{-}of\text{-}hterms\ [] = []$

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

theorem *herbrand-evalt* [*simp*]:
  closedt 0 t ⟹ *term-of-hterm* (*evalt e HApp t*) = *t*
  closedts 0 ts ⟹ *terms-of-hterms* (*evalts e HApp ts*) = *ts*
  ⟨*proof*⟩

theorem *herbrand-evalt′* [*simp*]:
  evalt e HApp (*term-of-hterm ht*) = *ht*
  evalts e HApp (*terms-of-hterms hts*) = *hts*
  ⟨*proof*⟩

theorem *closed-hterm* [*simp*]:
  closedt 0 (*term-of-hterm* (*ht*::′*a hterm*))
  closedts 0 (*terms-of-hterms* (*hts*::′*a hterm list*))
  ⟨*proof*⟩

theorem *measure-size-eq* [*simp*]: ((*x*, *y*) ∈ *measure f*) = (*f x* < *f y*)
  ⟨*proof*⟩

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* ⟹
  (*p* ∈ *H* ⟶ *closed 0 p* ⟶
    eval e HApp (λ*a ts*. *Pred a* (*terms-of-hterms ts*) ∈ *H*) *p*) ∧
  (*Neg p* ∈ *H* ⟶ *closed 0 p* ⟶
    eval e HApp (λ*a ts*. *Pred a* (*terms-of-hterms ts*) ∈ *H*) (*Neg p*))
  ⟨*proof*⟩

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*: ¬ *finite* (− (⋃ *p*∈*S*. *params p*))
  and *surj*: ∀ *y*. ∃ *n*. *y* = *f n*
  shows *alt-consistency C* ⟹ *S* ∈ *C* ⟹ *hintikka* (*Extend S C f*)
  ⟨*proof*⟩

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*:

$consistency\ C \Longrightarrow S \in C \Longrightarrow \neg\ finite\ (-\ (\bigcup p \in S.\ params\ p)) \Longrightarrow$
$p \in S \Longrightarrow closed\ 0\ p \Longrightarrow eval\ e\ HApp\ (\lambda a\ ts.$
$Pred\ a\ (terms\text{-}of\text{-}hterms\ ts) \in Extend\ S$
$(mk\text{-}finite\text{-}char\ (mk\text{-}alt\text{-}consistency\ (close\ C)))\ diag\text{-}form')\ p$
⟨*proof*⟩

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 ⊢ is a consistency property:

theorem *deriv-consistency*:
assumes *inf-param*: $\neg\ finite\ (UNIV::'a\ set)$
shows $consistency\ \{S::('a,\ 'b)\ form\ set.\ \exists\, G.\ S = set\ G \land \neg\ G \vdash FF\}$
⟨*proof*⟩

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

theorem *natded-complete*: $closed\ 0\ p \Longrightarrow list\text{-}all\ (closed\ 0)\ ps \Longrightarrow$
$\forall\, e\ (f::nat \Rightarrow nat\ hterm\ list \Rightarrow nat\ hterm)\ (g::nat \Rightarrow nat\ hterm\ list \Rightarrow bool).$
$e,f,g,ps \models p \Longrightarrow ps \vdash p$
⟨*proof*⟩

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*: $consistency\ \{S.\ \neg\ finite\ (-\ (\bigcup p{\in}S.\ params\ p)) \land$
$(\exists f.\ \forall (p::('a,\ 'b)form){\in}S.\ eval\ e\ f\ g\ p)\}$
⟨*proof*⟩

theorem *doublep-evalt* [*simp*]:
$evalt\ e\ f\ (psubstt\ (\lambda n::nat.\ 2 * n)\ t) = evalt\ e\ (\lambda n.\ f\ (2{*}n))\ t$
$evalts\ e\ f\ (psubstts\ (\lambda n::nat.\ 2 * n)\ ts) = evalts\ e\ (\lambda n.\ f\ (2{*}n))\ ts$
⟨*proof*⟩

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

theorem *doublep-infinite-params*:
$\neg\ finite\ (-\ (\bigcup p \in psubst\ (\lambda n::nat.\ 2 * n)\ `\ S.\ params\ p))$
⟨*proof*⟩

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*: $\forall\, p{\in}S.\ eval\ e\ f\ g\ p \Longrightarrow$
$\forall\, p{\in}S.\ closed\ 0\ p \longrightarrow eval\ e'\ (\lambda n.\ HApp\ (2{*}n))\ (\lambda a\ ts.$
$Pred\ a\ (terms\text{-}of\text{-}hterms\ ts) \in Extend\ (psubst\ (\lambda n.\ 2 * n)\ `\ S)$
$(mk\text{-}finite\text{-}char\ (mk\text{-}alt\text{-}consistency\ (close$
$\{S.\ \neg\ finite\ (-\ (\bigcup p{\in}S.\ params\ p)) \wedge$
$(\exists f.\ \forall\, p{\in}S.\ eval\ e\ f\ g\ p)\})))\ diag\text{-}form')\ p$
⟨*proof*⟩

References

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