Meta-theory of first-order predicate logic

Stefan Berghofer

August 28, 2014

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
1 Miscellaneous Utilities

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

**Theorem** conjE: \( P \land Q \Rightarrow (P \Rightarrow P') \land (Q \Rightarrow Q') \Rightarrow P' \land Q' \)

**Theorem** conjE": \( (\forall x. P x \Rightarrow Q x \land R x) \Rightarrow ((\forall x. P x \Rightarrow Q x) \Rightarrow Q') \Rightarrow ((\forall x. P x \Rightarrow R x) \Rightarrow R') \Rightarrow Q' \land R' \)

Some facts about (in)finite sets

**Theorem** [simp]: \( - A \cap B = B - A \) (proof)

**Theorem** Compl-UNIV-eq: \( - A = \text{UNIV} - A \) (proof)

**Theorem** infinite-nonempty: \( \neg \text{finite} A \Rightarrow \exists x. x \in A \) (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
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])
```
\[ [\emptyset][s/k] = \emptyset \]
\[ (t \# ts)[s/k] = t[s/k] \# ts[s/k] \]

\textbf{primrec}

\textit{liftt :: 'a term \Rightarrow 'a term}

\textit{and liftts :: 'a term list \Rightarrow 'a term list}

\textbf{where}

\textit{liftt (Var i) = Var (Suc i)}
\[ \textit{liftt (App a ts) = App a (liftts ts)} \]
\[ \textit{liftts [] = []} \]
\[ \textit{liftts (t \# ts) = liftt t \# liftts ts} \]

\textbf{primrec}

\textit{subst :: ('a, 'b) form \Rightarrow 'a term \Rightarrow nat \Rightarrow ('a, 'b) form} \((\cdot/\cdot) [300, 0, 0] 300)\]

\textbf{where}

\textit{FF[s/k] = FF}
\[ \textit{TT[s/k] = TT} \]
\[ \textit{(Pred b ts)[s/k] = Pred b (ts[s/k])} \]
\[ \textit{(And p q)[s/k] = And (p[s/k]) (q[s/k])} \]
\[ \textit{(Or p q)[s/k] = Or (p[s/k]) (q[s/k])} \]
\[ \textit{(Impl p q)[s/k] = Impl (p[s/k]) (q[s/k])} \]
\[ \textit{(Neg p)[s/k] = Neg (p[s/k])} \]
\[ \textit{(Forall p)[s/k] = Forall (p[liftt s/Suc k])} \]
\[ \textit{(Exists p)[s/k] = Exists (p[liftt s/Suc k])} \]

\textbf{theorem} \textit{lift-closed [simp]:}
\[ \textit{closedt 0 \ (t::'a term) \implies closedt 0 \ (liftt t)} \]
\[ \textit{closedts 0 \ (ts::'a term list) \implies closedts 0 \ (liftts ts)} \]

\textbf{⟨proof⟩}

\textbf{theorem} \textit{subst-closedt [simp]: assumes u: closedt 0 u}
\[ \textit{shows closedt (Suc i) t \implies closedt i \ (t[u/i])} \]
\[ \textit{and closedts (Suc i) ts \implies closedts i \ (ts[u/i])} \]

\textbf{⟨proof⟩}

\textbf{theorem} \textit{subst-closed [simp]:}
\[ \textit{closedt 0 \ t \implies closed \ (Suc i) \ p \implies closed \ i \ (p[t/i])} \]

\textbf{⟨proof⟩}

\textbf{theorem} \textit{subst-size [simp]: size (subst p t i) = size p}

\textbf{⟨proof⟩}

\textbf{2.3 Parameters}

The introduction rule \textit{ForallI} for the universal quantifier, as well as the elimination rule \textit{ExistsE} for the existential quantifier introduced in \S 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**

\[
\text{paramst :: 'a term ⇒ 'a set}
\]

and \(\text{paramsts :: 'a term list ⇒ 'a set}\)

**where**

\[
\text{paramst (Var n) = \{\}}
\]
\[
\text{paramst (App a ts) = \{a\} ∪ paramsts ts}
\]
\[
\text{paramsts [] = \{\}}
\]
\[
\text{paramsts (t # ts) = (paramst t ∪ paramsts ts)}
\]

**primrec**

\[
\text{params :: ('a, 'b) form ⇒ 'a set}
\]

**where**

\[
\text{params FF = \{\}}
\]
\[
\text{params TT = \{\}}
\]
\[
\text{params (Pred b ts) = paramsts ts}
\]
\[
\text{params (And p q) = params p ∪ params q}
\]
\[
\text{params (Or p q) = params p ∪ params q}
\]
\[
\text{params (Impl p q) = params p ∪ params q}
\]
\[
\text{params (Neg p) = params p}
\]
\[
\text{params (Forall p) = params p}
\]
\[
\text{params (Exists p) = params p}
\]

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

**primrec**

\[
\text{psubstt :: ('a ⇒ 'c) ⇒ 'a term ⇒ 'c term}
\]

and \(\text{psubstts :: ('a ⇒ 'c) ⇒ 'a term list ⇒ 'c term list}\)

**where**

\[
\text{psubstt f (Var i) = Var i}
\]
\[
\text{psubstt f (App x ts) = App (f x) (psubstts f ts)}
\]
\[
\text{psubstts f [] = []}
\]
\[
\text{psubstts f (t # ts) = psubst f t # psubstts f ts}
\]

**primrec**

\[
\text{psubst :: ('a ⇒ 'c) ⇒ ('a, 'b) form ⇒ ('c, 'b) form}
\]

**where**

\[
\text{psubst f FF = FF}
\]
\[
\text{psubst f TT = TT}
\]
\[
\text{psubst f (Pred b ts) = Pred b (psubstts f ts)}
\]
\[
\text{psubst f (And p q) = And (psubst f p) (psubst f q)}
\]
\[
\text{psubst f (Or p q) = Or (psubst f p) (psubst f q)}
\]
\[
\text{psubst f (Impl p q) = Impl (psubst f p) (psubst f q)}
\]
\[
\text{psubst f (Neg p) = Neg (psubst f p)}
\]
\[
\text{psubst f (Forall p) = Forall (psubst f p)}
\]
\[
\text{psubst f (Exists p) = Exists (psubst f p)}
\]

**theorem** \(\text{psubstt-closed [simp]}:
\)

\[
\text{closedt i (psubst f t) = closedt i t}
\]
\[ \text{closedts } i \ (\text{psubstts } f \ ts) = \text{closedts } i \ ts \]

**Theorem** \text{psubst-closed} [simp]:
\[ \text{closed } i \ (\text{psubst } f \ p) = \text{closed } i \ p \]

**Theorem** \text{psubstt-subst} [simp]:
\[ \text{psubstt } f \ (\text{substt } t \ u \ i) = \text{substt } (\text{psubstt } f \ t) \ (\text{psubstt } f \ u) \ i \]
\[ \text{psubstts } f \ (\text{substts } ts \ u \ i) = \text{substts } (\text{psubstts } f \ ts) \ (\text{psubstt } f \ u) \ i \]

**Theorem** \text{psubstt-lift} [simp]:
\[ \text{psubstt } f \ (\text{lifft } t) = \text{lifft } (\text{psubstt } f \ t) \]
\[ \text{psubstts } f \ (\text{liffts } ts) = \text{liffts } (\text{psubstts } f \ ts) \]

**Theorem** \text{psubst-subst} [simp]:
\[ \text{psubst } f \ (\text{subst } P \ t \ i) = \text{subst } (\text{psubst } f \ P) \ (\text{psubst } f \ t) \ i \]

**Theorem** \text{psubstt-upd} [simp]:
\[ x \not\in \text{paramst } (\text{t::'a term}) \implies \text{psubstt } (f(x:=y)) \ t = \text{psubstt } f \ t \]
\[ x \not\in \text{params } (\text{ts::'a term list}) \implies \text{psubstts } (f(x:=y)) \ ts = \text{psubstts } f \ ts \]

**Theorem** \text{psubst-upd} [simp]:
\[ x \not\in \text{params } P \implies \text{psubst } (f(x:=y)) \ P = \text{psubst } f \ P \]

**Theorem** \text{psubstt-id} [simp]:
\[ \text{psubstt } (\%x. \ x) \ (\text{t::'a term}) = t \]
\[ \text{psubstts } (\%x. \ x) \ (\text{ts::'a term list}) = ts \]

**Theorem** \text{psubst-id} [simp]:
\[ \text{psubst } (\%x. \ x) = (\%p. \ p) \]

**Theorem** \text{psubstt-image} [simp]:
\[ \text{paramst } (\text{psubst } f \ t) = f \cdot \text{paramst } t \]
\[ \text{params } (\text{psubstts } f \ ts) = f \cdot \text{params } ts \]

**Theorem** \text{psubst-image} [simp]:
\[ \text{params } (\text{psubst } f \ p) = f \cdot \text{params } p \]

### 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 \( e(i:a) \), 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

\[
\text{shift :: (nat \Rightarrow \text{'}a) \Rightarrow \text{'}a \Rightarrow \text{'}a \Rightarrow \text{'}a} \quad \text{((-\cdot\cdot)} [90, 0, 0] 91) \quad \text{where}
\]

\[
e(i:a) = (\lambda j. \text{if } j < i \text{ then } e \ j \text{ else if } j = i \text{ then } a \text{ else } e(j-1))
\]

lemma shift-eq [simp]: \( i = j \Rightarrow (e(i:T)) j = T \)
(proof)

lemma shift-gt [simp]: \( j < i \Rightarrow (e(i:T)) j = e \ j \)
(proof)

lemma shift-lt [simp]: \( i < j \Rightarrow (e(i:T)) j = e \ (j-1) \)
(proof)

lemma shift-commute [simp]: \( e(i:U)(\emptyset:T) = e(\emptyset:T)(\text{Suc } i:U) \)
(proof)

primrec

\[
evalt :: (\text{nat} \Rightarrow \text{'}c) \Rightarrow (\text{'}a \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}c) \Rightarrow \text{'}a \text{ term} \Rightarrow \text{'}c
\]

and evalts :: (\text{nat} \Rightarrow \text{'}c) \Rightarrow (\text{'}a \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}c) \Rightarrow \text{'}a \text{ term list} \Rightarrow \text{'}c \text{ list}

where

\[
evalt e f (\text{Var } n) = e n
\]
\[
\mid \text{evalt } e f (\text{App } a \ ts) = f a (\text{evalts } e f \ ts)
\]
\[
\mid \text{evalts } e f \ [] = []
\]
\[
\mid \text{evalts } e f (t \ # \ ts) = \text{evalt } e f t \ # \ \text{evalts } e f \ ts
\]

primrec

\[
eval :: (\text{nat} \Rightarrow \text{'}c) \Rightarrow (\text{'}a \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}c) \Rightarrow (\text{'}b \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}bool) \Rightarrow (\text{'}a, \text{'}b) \text{ form} \Rightarrow \text{'}bool
\]

where

\[
eval e f g \text{ } \text{FF} \ = \ False
\]
\[
\mid \text{eval } e f g \text{ } \text{TT} \ = \ True
\]
\[
\mid \text{eval } e f g \ (\text{Pred } a \ ts) = \text{ } g \ a \ (\text{evalts } e f \ ts)
\]
\[
\mid \text{eval } e f g \ (\text{And } p \ q) = ((\text{eval } e f g \ p) \ \land \ (\text{eval } e f g \ q))
\]
\[
\mid \text{eval } e f g \ (\text{Or } p \ q) = ((\text{eval } e f g \ p) \ \lor \ (\text{eval } e f g \ q))
\]
\[
\mid \text{eval } e f g \ (\text{Impl } p \ q) = ((\text{eval } e f g \ p) \ \rightarrow \ (\text{eval } e f g \ q))
\]
\[
\mid \text{eval } e f g \ (\text{Neg } p) = (\neg (\text{eval } e f g \ p))
\]
\[
\mid \text{eval } e f g \ (\text{Forall } p) = (\forall z. \text{ eval } (e(\emptyset:z)) f g \ p)
\]
\[
\mid \text{eval } e f g \ (\text{Exists } p) = (\exists z. \text{ eval } (e(\emptyset:z)) f g \ p)
\]

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

\[
\text{model :: (nat} \Rightarrow \text{'}c) \Rightarrow (\text{'}a \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}c) \Rightarrow (\text{'}b \Rightarrow \text{'}c \text{ list} \Rightarrow \text{'}bool) \Rightarrow
\]

7
(\textquote{a, b} \text{ form list} \Rightarrow (\textquote{a, b} \text{ form} \Rightarrow \text{ bool} \ (\cdot \cdot \cdot \vdash - [50,50] 50)) \ \textbf{where}
\textbf{where}
\textbf{where}
\begin{align*}
(e.f.g.ps \vdash p) &= (\text{list-all (eval e f g) ps} \rightarrow \text{eval e f g p})
\end{align*}

The following substitution lemmas relate substitution and evaluation functions:

\begin{theorem}
\textbf{subst-lemma'} [simp]:
\begin{align*}
evalt e f (\text{subst t u i}) &= \text{evalt} (e\langle i: \text{evalt e f u} \rangle) f t \\
\end{align*}
\end{theorem}

\begin{theorem}
\textbf{lift-lemma} [simp]:
\begin{align*}
evalt (e\langle 0: z \rangle) f (\text{lift t}) &= \text{evalt} e f t \\
\end{align*}
\end{theorem}

\begin{theorem}
\textbf{subst-lemma'} [simp]:
\begin{align*}
\forall e t. \text{eval e f g (subst a t i)} &= \text{eval} (e\langle i: \text{evalt e f t} \rangle) f g a
\end{align*}
\end{theorem}

\begin{theorem}
\textbf{upd-lemma'} [simp]:
\begin{align*}
n \notin \text{paramst t} \Rightarrow \text{eval e (f (n:=x)) t} &= \text{evalt} e f t \\
n \notin \text{paramsts ts} \Rightarrow \text{evalts e (f (n:=x)) ts} &= \text{evalts e f ts}
\end{align*}
\end{theorem}

\begin{theorem}
\textbf{upd-lemma} [simp]:
\begin{align*}
n \notin \text{params p} \Rightarrow \text{eval e (f (n:=x)) g p} &= \text{eval e f g p}
\end{align*}
\end{theorem}

\begin{theorem}
\textbf{list-upd-lemma} [simp]: list-all (\lambda p. n \notin \text{params p}) G \Rightarrow
\begin{align*}
\text{list-all (eval e (f (n:=x)) g) G} &= \text{list-all (eval e f g) G}
\end{align*}
\end{theorem}

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

\begin{theorem}
\textbf{ex-all-commute-eval}:
\begin{align*}
\text{eval e f g (Impl (Exists (Forall (Pred p [Var 1, Var 0]))) (Forall (Exists (Pred p [Var 0, Var 1])))})
\end{align*}
\end{theorem}

\section{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.

\begin{inductive}
\begin{align*}
\text{deriv :: ('a, 'b) form list} \Rightarrow ('a, 'b) form \Rightarrow \text{ bool} \ (\cdot \vdash - [50,50] 50)
\end{align*}
\end{inductive}

\begin{where}
\textbf{Assum}: a \in \text{ set G} \Rightarrow G \vdash a
\end{where}
The following derived inference rules are sometimes useful in applications.

theorem cut: \( G \vdash A \Rightarrow A \# G \vdash B \Rightarrow G \vdash B \)  

(proof)

theorem cut': \( A \# G \vdash B \Rightarrow G \vdash A \Rightarrow G \vdash B \)  

(proof)

theorem Class': \( \text{Neg} A \# G \vdash A \Rightarrow G \vdash A \)  

(proof)

theorem ForallE': \( G \vdash \text{Forall} a \Rightarrow \text{subst} a t 0 \# G \vdash B \Rightarrow G \vdash B \)  

(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 \text{Or} (\text{Pred} p []) (\text{Neg} (\text{Pred} p []))  

(proof)

theorem ex-all-commute:  
\( ([]::(\text{nat}, 'b) \text{ form list}) \vdash \text{Impl} (\text{Exists} (\text{Forall} (\text{Pred} p [\text{Var} 1, \text{Var} 0]))) \)  
\( (\text{Forall} (\text{Exists} (\text{Pred} p [\text{Var} 0, \text{Var} 1]))) \)  

(proof)

theorem drinker: ([])::(\text{nat}, 'b) \text{ form list} \vdash \text{Exists} (\text{Impl} (\text{Pred} p [\text{Var} 0]) (\text{Forall} (\text{Pred} p [\text{Var} 0])))  

(proof)

theorem peirce:
⊢ Impl (Impl (Impl (Pred P [])) (Pred Q [])) (Pred P [])) (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:** \( G \vdash p \implies \forall e f g. e, f, g, G \models p \)

⟨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. \( ps \models p \implies ps \vdash p \)
2. All consistent sets are satisfiable

The latter property is called the model existence theorem. To see why 2 implies 1, observe that \( \neg p, ps \not\vdash FF \) 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. \( S \not\vdash FF \). 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 \( \phi_0, \phi_1, \phi_2, \ldots \) 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**

\[ \text{consistency} :: (\forall p. ts. \neg (\text{Pred } p \ ts \in S \land \neg \text{Pred } p \ ts) \in S) \land \]
\[ FF \notin S \land \neg \text{TT} \notin S \land \]
\[ (\forall Z. \neg (\neg \text{Z} \in S \rightarrow S \cup \{Z\} \in C) \land \]
\[ (\forall A. B. \text{And } A B \in S \rightarrow S \cup \{A, B\} \in C) \land \]
\[ (\forall A. B. \neg (\text{Or } A B \in S \rightarrow S \cup \{\neg A, \neg B\} \in C) \land \]
\[ (\forall A. B. \text{Impl } A B \in S \rightarrow S \cup \{\neg A\} \in C \land S \cup \{B\} \in C) \land \]
\[ (\forall A. B. \neg (\text{Impl } A B \in S \rightarrow S \cup \{\neg A\} \in C) \land \]
\[ (\forall P. \text{Exists } P \in S \rightarrow (\exists x. S \cup \{P[\text{App } x \ [0/0]] \in C) \land \]
\[ (\forall P. \neg (\text{Forall } P \in S \rightarrow S \cup \{P[t/0]\} \in C) \land \]

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**

\[ \text{alt-consistency} :: (\forall p. ts. \neg (\text{Pred } p \ ts \in S \land \neg \text{Pred } p \ ts) \in S) \land \]
\[ FF \notin S \land \neg \text{TT} \notin S \land \]
\[ (\forall Z. \neg (\neg \text{Z} \in S \rightarrow S \cup \{Z\} \in C) \land \]
\[ (\forall A. B. \text{And } A B \in S \rightarrow S \cup \{A, B\} \in C) \land \]
\[ (\forall A. B. \neg (\text{Or } A B \in S \rightarrow S \cup \{\neg A, \neg B\} \in C) \land \]
\[ (\forall A. B. \text{Impl } A B \in S \rightarrow S \cup \{\neg A\} \in C \land S \cup \{B\} \in C) \land \]
\[ (\forall A. B. \neg (\text{Impl } A B \in S \rightarrow S \cup \{\neg A\} \in C) \land \]
\[ (\forall P. \text{Exists } P \in S \rightarrow (\exists x. S \cup \{P[\text{App } x \ [0/0]] \in C) \land \]
\[ (\forall P. \neg (\text{Forall } P \in S \rightarrow S \cup \{P[t/0]\} \in C) \land \]

11
\[ S \cup \{ \text{Neg} \ (P[\text{App} \ x \ \emptyset]) \} \in C) \]

Note that in the clauses for \( \exists P \) and \( \text{Neg} \ (\forall P) \), 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**

\[
\text{mk-alt-consistency} :: (\ 'a, 'b) \text{ form set set} \Rightarrow (\ 'a, 'b) \text{ form set set}
\]

\[
\text{mk-alt-consistency} \ C = \{ S. \ \exists f. \ \text{psubst} \ f \ S \in C \}
\]

**theorem** \( \text{alt-consistency} \):

\[
\text{consistency} \ C = \Rightarrow \text{alt-consistency} \ (\text{mk-alt-consistency} \ C)
\]

**proof**

**definition**

\[
\text{close} :: (\ 'a, 'b) \text{ form set set} \Rightarrow (\ 'a, 'b) \text{ form set set}
\]

\[
\text{close} \ C = \{ S. \ \exists S' \in C. \ S \subseteq S' \}
\]

**definition**

\[
\text{subset-closed} :: \ 'a \text{ set set} \Rightarrow \text{bool}
\]

\[
\text{subset-closed} \ C = \{ \forall S' \in C. \forall S. \ S \subseteq S' \Rightarrow S \in C \}
\]

**theorem** \( \text{close-consistency} \):

\[
\text{consistency} \ C = \Rightarrow \text{consistency} \ (\text{close} \ C)
\]

**proof**

**theorem** \( \text{close-closed} \):

\[
\text{subset-closed} \ (\text{close} \ C)
\]

**proof**

**theorem** \( \text{close-subset} \):

\[
C \subseteq \text{close} \ C
\]

**proof**

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

**theorem** \( \text{mk-alt-consistency-closed} \):

\[
\text{subset-closed} \ C = \Rightarrow \text{subset-closed} \ (\text{mk-alt-consistency} \ C)
\]

**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**

\[
\text{finite-char} :: \forall \text{ set set} \Rightarrow \text{ bool where } \\
\text{finite-char} \ C = (\forall S. S \in C = (\forall S'. \text{ finite } S' \rightarrow S' \subseteq S \rightarrow S' \in C))
\]

**definition**

\[
\text{mk-finite-char} :: \forall \text{ set set} \Rightarrow \forall \text{ set set where } \\
\text{mk-finite-char} \ C = \{ S. \forall S'. S' \subseteq S \Rightarrow \text{ finite } S' \Rightarrow S' \in C\}
\]

**theorem** finite-alt-consistency:

\[\text{alt-consistency} \ C \Rightarrow \text{ subset-closed} \ C \Rightarrow \text{ alt-consistency} (\text{mk-finite-char} \ C)\]

(\text{proof})

**theorem** finite-char: finite-char (\text{mk-finite-char} \ C)

(\text{proof})

**theorem** finite-char-closed: finite-char \ C \Rightarrow \text{ subset-closed} \ C

(\text{proof})

**theorem** finite-char-subset: subset-closed \ C \Rightarrow \text{ C \subseteq mk-finite-char} \ C

(\text{proof})

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**

\[
\text{diag} :: \text{ nat} \Rightarrow (\text{ nat } \times \text{ nat})
\]

**where**

\[
\text{diag} \ 0 = (0, 0) \\
\text{diag} (\text{ Suc } n) = \\
\ \\
\text{ (let } (x, y) = \text{ diag } n \\
\text{ in case } y \text{ of } \\
\ \\
\text{ 0 } \Rightarrow (0, \text{ Suc } x)
\]
| Suc y ⇒ (Suc x, y))

**theorem** diag-le1: \( \text{fst} \ (\text{diag} \ (\text{Suc} \ n)) < \text{Suc} \ n \)  
(proof)

**theorem** diag-le2: \( \text{snd} \ (\text{diag} \ (\text{Suc} \ (\text{Suc} \ n))) < \text{Suc} \ (\text{Suc} \ n) \)  
(proof)

**theorem** diag-le3: \( \text{fst} \ (\text{diag} \ n) = \text{Suc} \ x \Rightarrow \text{snd} \ (\text{diag} \ n) < n \)  
(proof)

**theorem** diag-le4: \( \text{fst} \ (\text{diag} \ n) = \text{Suc} \ x \Rightarrow x < n \)  
(proof)

**function** undiag :: nat × nat ⇒ nat  
**where**  
undiag \( (0, 0) = 0 \)  
| undiag \( (0, \text{Suc} \ y) = \text{Suc} \ (\text{undiag} \ (y, 0)) \)  
| undiag \( (\text{Suc} \ x, y) = \text{Suc} \ (\text{undiag} \ (x, \text{Suc} \ y)) \)  
(proof)

**termination**  
(proof)

**theorem** diag-undiag [simp]: \( \text{diag} \ (\text{undiag} \ (x, y)) = (x, y) \)  
(proof)
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 [] = Leaf 0
btree-of-list (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]:
(∀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: ∀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]:
(∀x. d (u x) = x) ⇒ diag-term d (undiag-term u t) = t
(proof)

fun
\[
\text{form-of-btree} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'b) \Rightarrow \text{btree} \Rightarrow ('a, 'b) \text{ form}
\]

\begin{align*}
\text{form-of-btree} & \cdot f \ g \ (\text{Leaf} \ 0) \ = \ \text{FF} \\
\text{form-of-btree} & \cdot f \ g \ (\text{Leaf} \ (\text{Suc} \ 0)) \ = \ \text{TT} \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ 0) \ (\text{Branch} \ (\text{Leaf} \ m) \ (\text{Leaf} \ n))) = \\
& \quad \quad \quad \quad \quad \text{Pred} \ (g \ m) \ (\text{diag-list} \ (\text{diag-term} \ f) \ n) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ 0)) \ (\text{Branch} \ t1 \ t2)) = \\
& \quad \quad \quad \quad \quad \text{And} \ (\text{form-of-btree} \ f \ g \ t1) \ (\text{form-of-btree} \ f \ g \ t2) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ 0))) \ (\text{Branch} \ t1 \ t2)) = \\
& \quad \quad \quad \quad \quad \text{Or} \ (\text{form-of-btree} \ f \ g \ t1) \ (\text{form-of-btree} \ f \ g \ t2) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ 0)))) \ (\text{Branch} \ t1 \ t2)) = \\
& \quad \quad \quad \quad \quad \text{Impl} \ (\text{form-of-btree} \ f \ g \ t1) \ (\text{form-of-btree} \ f \ g \ t2) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ 0)))) \ t) = \\
& \quad \quad \quad \quad \quad \text{Neg} \ (\text{form-of-btree} \ f \ g \ t) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ 0)))))) \ t) = \\
& \quad \quad \quad \quad \quad \text{Forall} \ (\text{form-of-btree} \ f \ g \ t) \\
\text{form-of-btree} & \cdot f \ g \ (\text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ 0)))))) \ t) = \\
& \quad \quad \quad \quad \quad \text{Exists} \ (\text{form-of-btree} \ f \ g \ t)
\end{align*}

\textbf{primrec}
\[
\text{btree-of-form} :: ('a \Rightarrow \text{nat}) \Rightarrow ('b \Rightarrow \text{nat}) \Rightarrow ('a, 'b) \text{ form} \Rightarrow \text{btree}
\]

\begin{align*}
\text{btree-of-form} & \cdot f \ g \ \text{FF} = \text{Leaf} \ 0 \\
\text{btree-of-form} & \cdot f \ g \ \text{TT} = \text{Leaf} \ (\text{Suc} \ 0) \\
\text{branch-of-form} & \cdot f \ g \ (\text{Pred} \ b \ ts) = \text{Branch} \ (\text{Leaf} \ 0) \\
& \quad \quad \quad \quad \quad (\text{Branch} \ (\text{Leaf} \ (g \ b)) \ (\text{Leaf} \ (\text{undiaform} \ (\text{undiaform} \ f) \ ts))) \\
\text{btree-of-form} & \cdot f \ g \ (\text{And} \ a \ b) = \text{Branch} \ (\text{Leaf} \ (\text{Suc} \ 0)) \\
& \quad \quad \quad \quad \quad (\text{Branch} \ (\text{btree-of-form} \ f \ g \ a) \ (\text{btree-of-form} \ f \ g \ b)) \\
\text{btree-of-form} & \cdot f \ g \ (\text{Or} \ a \ b) = \text{Branch} \ (\text{Leaf} \ (\text{Suc} \ 0)) \\
& \quad \quad \quad \quad \quad (\text{Branch} \ (\text{btree-of-form} \ f \ g \ a) \ (\text{btree-of-form} \ f \ g \ b)) \\
\text{btree-of-form} & \cdot f \ g \ (\text{Impl} \ a \ b) = \text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ 0))) \\
& \quad \quad \quad \quad \quad (\text{Branch} \ (\text{btree-of-form} \ f \ g \ a) \ (\text{btree-of-form} \ f \ g \ b)) \\
\text{btree-of-form} & \cdot f \ g \ (\text{Neg} \ a) = \text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ 0)))) \\
& \quad \quad \quad \quad \quad (\text{btree-of-form} \ f \ g \ a) \\
\text{btree-of-form} & \cdot f \ g \ (\text{Forall} \ a) = \text{Branch} \ (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ 0))))))) \\
& \quad \quad \quad \quad \quad (\text{btree-of-form} \ f \ g \ a) \\
\text{btree-of-form} & \cdot f \ g \ (\text{Exists} \ a) = \text{Branch} \\
& \quad \quad \quad \quad \quad (\text{Leaf} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ (\text{Suc} \ 0))))))) \\
& \quad \quad \quad \quad \quad (\text{btree-of-form} \ f \ g \ a)
\end{align*}

\textbf{definition}
\[
\text{diag-form} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'b) \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ form}
\]

\textbf{where}
\[
\text{diag-form} \ f \ g \ n = \text{form-of-btree} \ f \ g \ (\text{diag-btree} \ n)
\]

\textbf{definition}
\[
\text{undiaform} :: ('a \Rightarrow \text{nat}) \Rightarrow ('b \Rightarrow \text{nat}) \Rightarrow ('a, 'b) \text{ form} \Rightarrow \text{nat}
\]

\textbf{where}
\[
\text{undiaform} \ f \ g \ x = \text{undiaform} \ (\text{btree-of-form} \ f \ g \ x)
\]

\textbf{theorem}
\[
\text{diag-undiaform} \ [\text{simp}]:
\]
\[(\forall x. d (u x) = x) \implies (\forall x. d' (u' x) = x) \implies \text{diag-form } d d' (\text{undiag-form } u u' f) = f\]

**definition**

\[\text{diag-form}' :: \text{nat} \Rightarrow (\text{nat}, \text{nat}) \text{ form where} \]

\[\text{diag-form}' = \text{diag-form } (\lambda n. n) (\lambda n. n)\]

**definition**

\[\text{undiag-form}' :: (\text{nat}, \text{nat}) \text{ form } \Rightarrow \text{nat where} \]

\[\text{undiag-form}' = \text{undiag-form } (\lambda n. n) (\lambda n. n)\]

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

### 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**

\[\text{is-chain} :: (\text{nat} \Rightarrow 'a \text{ set}) \Rightarrow \text{bool where} \]

\[\text{is-chain } f = (\forall n. f n \subseteq f (\text{Suc } n))\]

**theorem** \[\text{is-chainD}: \text{is-chain } f \implies x \in f m \implies x \in f (m + n)\]

**theorem** \[\text{is-chainD'}: \text{is-chain } f \implies x \in f m \implies m \leq k \implies x \in f k\]

**theorem** \[\text{chain-index}: \]

\[\text{assumes } ch: \text{is-chain } f \text{ and } \text{fin: finite } F \]

\[\text{shows } F \subseteq (\bigcup n. f n) \implies \exists n. F \subseteq f n \text{ (proof)}\]

**theorem** \[\text{chain-union-closed}: \]

\[\text{finite-char } C \implies \text{is-chain } f \implies \forall n. f n \in C \implies (\bigcup n. f n) \in C \text{ (proof)}\]

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

**primrec**

\[\text{dest-Neg} :: ('a, 'b) \text{ form } \Rightarrow ('a, 'b) \text{ form where} \]

\[\text{dest-Neg } (\text{Neg } p) = p\]

**primrec**

\[\text{dest-Forall} :: ('a, 'b) \text{ form } \Rightarrow ('a, 'b) \text{ form where} \]
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))
(Appl (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))))
(Appl (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: all-consistency $C \implies$ finite-char $C \implies$
$S \in C \implies \neg$ finite $(\bigcup p \in S. \text{params } p)) \implies \text{Extend } S \in C f \in C$
(proof)

theorem Extend-subset: $S \subseteq \text{Extend } S \subseteq C f$
(proof)

The Extend function yields a maximal set:

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

theorem extend-maximal: $\forall y. \exists n. y = f n \implies$
finite-char $C \implies$ maximal $(\text{Extend } S \subseteq C f) C$
(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 $H =$
$(\forall p \ ts. \neg (\text{Pred } p \ ts \in H \land \neg (\text{Pred } p \ ts) \in H)) \land$
$\neg FF \notin H \land \neg TT \notin H \land$
$(\forall Z. \neg (\text{Neg } Z) \in H \implies Z \in H) \land$
$(\forall A B. \text{And } A B \in H \implies A \in H \land B \in H) \land$
$(\forall A B. \neg (\text{Or } A B) \in H \implies \neg (\text{Neg } A \land \neg B) \in H) \land$
$(\forall A B. \text{Impl } A B \in H \implies \neg (\text{Neg } A \lor B) \in H) \land$
$(\forall A B. \neg (\text{Impl } A B) \in H \implies \neg (\text{Neg } A \lor B) \in H) \land$
$(\forall P. \text{Exists } P \in H \implies (\exists t. \text{closedt } 0 t \land \neg \text{subst } P t 0) \in H) \land$
$(\forall P. \text{Forall } P \in H \implies (\forall t. \text{closedt } 0 t \land \neg \text{subst } P t 0) \in H) \land$
$(\forall P. \neg (\text{Forall } P) \in H \implies (\exists t. \text{closedt } 0 t \land \neg (\text{subst } P t 0) \in H))$

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-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]:**

\[
\text{closedt } 0 \ t \implies \text{term-of-hterm } (\text{evalt } e \ \text{HApp } t) = t
\]

\[
\text{closedts } 0 \ ts \implies \text{terms-of-hterms } (\text{evalts } e \ \text{HApp } ts) = ts
\]

(Proof)

**Theorem herbrand-evalt' [simp]:**

\[
\text{evalt } e \ \text{HApp } (\text{term-of-hterm } ht) = ht
\]

\[
\text{evalts } e \ \text{HApp } (\text{terms-of-hterms } hts) = hts
\]

(Proof)

**Theorem closed-hterm [simp]:**

\[
\text{closedt } 0 \ (\text{term-of-hterm } (ht::'a \ \text{hterm}))
\]

\[
\text{closedts } 0 \ (\text{terms-of-hterms } (hts::'a \ \text{hterm list}))
\]

(Proof)

**Theorem measure-size-eq [simp]:**

\[
((x, y) \in \text{measure } f) = (f \ x < f \ y)
\]

(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 \(H \implies\)**

\[
(p \in H \implies \text{closed } 0 \ p \implies \\
\text{eval } e \ \text{HApp } (\lambda a \ ts. \ \text{Pred } a \ (\text{terms-of-hterms } ts) \in H) \ p \wedge \\
(Neg \ p \in H \implies \text{closed } 0 \ p \implies \\
\text{eval } e \ \text{HApp } (\lambda a \ ts. \ \text{Pred } a \ (\text{terms-of-hterms } ts) \in H) \ (Neg \ p))
\]

(Proof)

Using the maximality of \(\text{Extend } S \ C \ f\), we can show that \(\text{Extend } S \ C \ f\) yields Hintikka sets:

**Theorem extend-hintikka:**

**assumes** fin-ch: finite-char \(C\)

**and** infin-p: ¬ finite (¬ (\(\bigcup \text{p} \in S. \ \text{params } p\)))

**and** surj: \(\forall y. \exists n. \ y = f \ n\)

**shows** alt-consistency \(C \implies S \in C \implies \text{hintikka } (\text{Extend } S \ C \ f)\)

(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 $C \implies S \in C \implies \neg \text{finite } (- (\bigcup p \in S. \text{params } p)) \implies$
p $p \in S \implies \text{closed } 0 p \implies \text{eval } e \text{HAapp } (\lambda a \text{ts}.
(\text{mk-finite-char } (\text{mk-all-consistency } (\text{close } C))) \text{ diag-form'} ) p$

\section*{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:

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

\langle proof \rangle

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

\textbf{theorem natded-complete:}
$\forall e, f, g, ps | p = \neg \text{finite } (\bigcup p \in \text{subst} (\lambda n::\text{nat}. 2 * n) \text{params } p) \implies$
$\forall e, f, g, ps | p = \text{eval } e f g p$

\langle proof \rangle
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.

\textbf{theorem} loewenheim-skolem: $\forall p \in S. \eval e f g p \implies$
$
\forall p \in S. \closed 0 p \implies \eval e' (\lambda n. \HApp (2* n)) (\lambda a ts.$
$\Pred a (\text{terms-of-hterms ts}) \in \text{Extend (psubst (\lambda n \cdot 2 * n) ' S)}$
$(\text{mk-finite-char (mk-alt-consistency (close}$
$\{S. \neg \text{finite } (\bigcup p \in S. \text{params } p) \} \land$
$(\exists f. \forall p \in S. \eval e f g p))) \text{ diag-form'}) p$

\textit{proof}

\textbf{References}