

Meta-theory of first-order predicate logic

Stefan Berghofer

May 14, 2024

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	First-Order Logic According to Fitting	2
2	Miscellaneous Utilities	2
3	Terms and formulae	2
3.1	Closed terms and formulae	3
3.2	Substitution	3
3.3	Parameters	5
4	Semantics	7
5	Proof calculus	9
6	Correctness	10
7	Completeness	10
7.1	Consistent sets	11
7.2	Closure under subsets	12
7.3	Finite character	13
7.4	Enumerating datatypes	13
7.5	Extension to maximal consistent sets	14
7.6	Hintikka sets and Herbrand models	16
7.7	Model existence theorem	18
7.8	Completeness for Natural Deduction	19

8	Löwenheim-Skolem theorem	19
9	Completeness for open formulas	20
9.1	Renaming	20
9.2	Substitution for constants	20
9.3	Weakening assumptions	24
9.4	Implications and assumptions	24
9.5	Closure elimination	25
9.6	Completeness	27

1 First-Order Logic According to Fitting

2 Miscellaneous Utilities

Some facts about (in)finite sets

theorem *set-inter-compl-diff [simp]*: $\langle - A \cap B = B - A \rangle$ *⟨proof⟩*

3 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.

The size of a formula is used later for wellfounded induction. The default implementation provided by the datatype package is not quite what we need, so here is an alternative version:

```
primrec size-form :: ⟨('a, 'b) form ⇒ nat⟩ where
```

$\langle \text{size-form } FF = 0 \rangle$
 $\mid \langle \text{size-form } TT = 0 \rangle$
 $\mid \langle \text{size-form } (\text{Pred } -) = 0 \rangle$
 $\mid \langle \text{size-form } (\text{And } p \ q) = \text{size-form } p + \text{size-form } q + 1 \rangle$
 $\mid \langle \text{size-form } (\text{Or } p \ q) = \text{size-form } p + \text{size-form } q + 1 \rangle$
 $\mid \langle \text{size-form } (\text{Impl } p \ q) = \text{size-form } p + \text{size-form } q + 1 \rangle$
 $\mid \langle \text{size-form } (\text{Neg } p) = \text{size-form } p + 1 \rangle$
 $\mid \langle \text{size-form } (\text{Forall } p) = \text{size-form } p + 1 \rangle$
 $\mid \langle \text{size-form } (\text{Exists } p) = \text{size-form } p + 1 \rangle$

3.1 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

$\text{closedt} :: \langle \text{nat} \Rightarrow 'a \ \text{term} \Rightarrow \text{bool} \rangle$ **and**
 $\text{closedts} :: \langle \text{nat} \Rightarrow 'a \ \text{term list} \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{closedt } m \ (\text{Var } n) = (n < m) \rangle$
 $\mid \langle \text{closedt } m \ (\text{App } a \ ts) = \text{closedts } m \ ts \rangle$
 $\mid \langle \text{closedts } m \ [] = \text{True} \rangle$
 $\mid \langle \text{closedts } m \ (t \ # \ ts) = (\text{closedt } m \ t \wedge \text{closedts } m \ ts) \rangle$

primrec closed :: $\langle \text{nat} \Rightarrow ('a, 'b) \ \text{form} \Rightarrow \text{bool} \rangle$ where

$\langle \text{closed } m \ FF = \text{True} \rangle$
 $\mid \langle \text{closed } m \ TT = \text{True} \rangle$
 $\mid \langle \text{closed } m \ (\text{Pred } b \ ts) = \text{closedts } m \ ts \rangle$
 $\mid \langle \text{closed } m \ (\text{And } p \ q) = (\text{closed } m \ p \wedge \text{closed } m \ q) \rangle$
 $\mid \langle \text{closed } m \ (\text{Or } p \ q) = (\text{closed } m \ p \wedge \text{closed } m \ q) \rangle$
 $\mid \langle \text{closed } m \ (\text{Impl } p \ q) = (\text{closed } m \ p \wedge \text{closed } m \ q) \rangle$
 $\mid \langle \text{closed } m \ (\text{Neg } p) = \text{closed } m \ p \rangle$
 $\mid \langle \text{closed } m \ (\text{Forall } p) = \text{closed } (\text{Suc } m) \ p \rangle$
 $\mid \langle \text{closed } m \ (\text{Exists } p) = \text{closed } (\text{Suc } m) \ p \rangle$

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

shows $\langle \text{closedt } i \ (t :: 'a \ \text{term}) \implies \text{closedt } j \ t \rangle$
and $\langle \text{closedts } i \ (ts :: 'a \ \text{term list}) \implies \text{closedts } j \ ts \rangle$
 $\langle \text{proof} \rangle$

theorem closed-mono: assumes $le: \langle i \leq j \rangle$

shows $\langle \text{closed } i \ p \implies \text{closed } j \ p \rangle$
 $\langle \text{proof} \rangle$

3.2 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

$subst :: \langle 'a \text{ term} \Rightarrow 'a \text{ term} \Rightarrow \text{nat} \Rightarrow 'a \text{ term} \rangle (-[-'/-] [300, 0, 0] 300)$ **and**
 $substts :: \langle 'a \text{ term list} \Rightarrow 'a \text{ term} \Rightarrow \text{nat} \Rightarrow 'a \text{ term list} \rangle (-[-'/-] [300, 0, 0] 300)$

where

$\langle (\text{Var } i)[s/k] = (\text{if } k < i \text{ then } \text{Var } (i - 1) \text{ else if } i = k \text{ then } s \text{ else } \text{Var } i) \rangle$
 $| \langle (\text{App } a \ ts)[s/k] = \text{App } a \ (ts[s/k]) \rangle$
 $| \langle [][s/k] = [] \rangle$
 $| \langle (t \ # \ ts)[s/k] = t[s/k] \ # \ ts[s/k] \rangle$

primrec

$liftt :: \langle 'a \text{ term} \Rightarrow 'a \text{ term} \rangle$ **and**
 $liftts :: \langle 'a \text{ term list} \Rightarrow 'a \text{ term list} \rangle$ **where**
 $\langle liftt (\text{Var } i) = \text{Var } (\text{Suc } i) \rangle$
 $| \langle liftt (\text{App } a \ ts) = \text{App } a \ (liftts \ ts) \rangle$
 $| \langle liftts [] = [] \rangle$
 $| \langle liftts (t \ # \ ts) = liftt \ t \ # \ liftts \ ts \rangle$

primrec $subst :: \langle ('a, 'b) \text{ form} \Rightarrow 'a \text{ term} \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ form} \rangle$

$(-[-'/-] [300, 0, 0] 300)$ **where**
 $\langle FF[s/k] = FF \rangle$
 $| \langle TT[s/k] = TT \rangle$
 $| \langle (\text{Pred } b \ ts)[s/k] = \text{Pred } b \ (ts[s/k]) \rangle$
 $| \langle (\text{And } p \ q)[s/k] = \text{And } (p[s/k]) \ (q[s/k]) \rangle$
 $| \langle (\text{Or } p \ q)[s/k] = \text{Or } (p[s/k]) \ (q[s/k]) \rangle$
 $| \langle (\text{Impl } p \ q)[s/k] = \text{Impl } (p[s/k]) \ (q[s/k]) \rangle$
 $| \langle (\text{Neg } p)[s/k] = \text{Neg } (p[s/k]) \rangle$
 $| \langle (\text{Forall } p)[s/k] = \text{Forall } (p[liftt \ s/\text{Suc } k]) \rangle$
 $| \langle (\text{Exists } p)[s/k] = \text{Exists } (p[liftt \ s/\text{Suc } k]) \rangle$

theorem *lift-closed* [simp]:

$\langle \text{closedt } 0 \ (t :: 'a \text{ term}) \Longrightarrow \text{closedt } 0 \ (liftt \ t) \rangle$
 $\langle \text{closedts } 0 \ (ts :: 'a \text{ term list}) \Longrightarrow \text{closedts } 0 \ (liftts \ ts) \rangle$
 $\langle \text{proof} \rangle$

theorem *subst-closedt* [simp]:

assumes $u: \langle \text{closedt } 0 \ u \rangle$
shows $\langle \text{closedt } (\text{Suc } i) \ t \Longrightarrow \text{closedt } i \ (t[u/i]) \rangle$
and $\langle \text{closedts } (\text{Suc } i) \ ts \Longrightarrow \text{closedts } i \ (ts[u/i]) \rangle$
 $\langle \text{proof} \rangle$

theorem *subst-closed* [simp]:

$\langle \text{closedt } 0 \ t \Longrightarrow \text{closed } (\text{Suc } i) \ p \Longrightarrow \text{closed } i \ (p[t/i]) \rangle$
 $\langle \text{proof} \rangle$

theorem *subst-size-form* [simp]: $\langle \text{size-form } (\text{subst } p \ t \ i) = \text{size-form } p \rangle$

$\langle \text{proof} \rangle$

3.3 Parameters

The introduction rule *ForallI* for the universal quantifier, as well as the elimination rule *ExistsE* for the existential quantifier introduced in §5 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

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

primrec

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

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

primrec

$psubst :: \langle ('a \Rightarrow 'c) \Rightarrow 'a \text{ term} \Rightarrow 'c \text{ term} \rangle$ **and**
 $psubstts :: \langle ('a \Rightarrow 'c) \Rightarrow 'a \text{ term list} \Rightarrow 'c \text{ term list} \rangle$ **where**
 $\langle psubst \text{ f (Var } i) = \text{Var } i \rangle$
 $\langle psubst \text{ f (App } x \text{ ts)} = \text{App } (f \text{ x}) (psubstts \text{ f ts}) \rangle$
 $\langle psubstts \text{ f []} = \text{[]} \rangle$
 $\langle psubstts \text{ f (t \# ts)} = psubst \text{ f t \# psubstts \text{ f ts} \rangle$

primrec

$psubst :: \langle ('a \Rightarrow 'c) \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('c, 'b) \text{ form} \rangle$ **where**
 $\langle psubst \text{ f FF} = \text{FF} \rangle$
 $\langle psubst \text{ f TT} = \text{TT} \rangle$
 $\langle psubst \text{ f (Pred } b \text{ ts)} = \text{Pred } b \text{ (psubstts \text{ f ts})} \rangle$
 $\langle psubst \text{ f (And } p \text{ q)} = \text{And } (psubst \text{ f p}) (psubst \text{ f q}) \rangle$
 $\langle psubst \text{ f (Or } p \text{ q)} = \text{Or } (psubst \text{ f p}) (psubst \text{ f q}) \rangle$
 $\langle psubst \text{ f (Impl } p \text{ q)} = \text{Impl } (psubst \text{ f p}) (psubst \text{ f q}) \rangle$
 $\langle psubst \text{ f (Neg } p) = \text{Neg } (psubst \text{ f p}) \rangle$
 $\langle psubst \text{ f (Forall } p) = \text{Forall } (psubst \text{ f p}) \rangle$
 $\langle psubst \text{ f (Exists } p) = \text{Exists } (psubst \text{ f p}) \rangle$

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

$\langle \text{closedt } i \text{ (psubstt } f \ t) = \text{closedt } i \ t \rangle$
 $\langle \text{closedts } i \text{ (psubstts } f \ ts) = \text{closedts } i \ ts \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \text{closed } i \text{ (psubst } f \ p) = \text{closed } i \ p \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \text{psubstt } f \text{ (substt } t \ u \ i) = \text{substt (psubstt } f \ t) \text{ (psubstt } f \ u) \ i \rangle$
 $\langle \text{psubstts } f \text{ (substts } ts \ u \ i) = \text{substts (psubstts } f \ ts) \text{ (psubstt } f \ u) \ i \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \text{psubstt } f \text{ (liftt } t) = \text{liftt (psubstt } f \ t) \rangle$
 $\langle \text{psubstts } f \text{ (liftts } ts) = \text{liftts (psubstts } f \ ts) \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \text{psubst } f \text{ (subst } P \ t \ i) = \text{subst (psubst } f \ P) \text{ (psubstt } f \ t) \ i \rangle$
 $\langle \text{proof} \rangle$

theorem *psubstt-upd* [*simp*]:

$\langle x \notin \text{paramst } (t :: 'a \text{ term}) \implies \text{psubstt } (f(x := y)) \ t = \text{psubstt } f \ t \rangle$
 $\langle x \notin \text{paramsts } (ts :: 'a \text{ term list}) \implies \text{psubstts } (f(x := y)) \ ts = \text{psubstts } f \ ts \rangle$
 $\langle \text{proof} \rangle$

theorem *psubst-upd* [*simp*]: $\langle x \notin \text{params } P \implies \text{psubst } (f(x := y)) \ P = \text{psubst } f \ P \rangle$

$\langle \text{proof} \rangle$

theorem *psubstt-id*:

fixes $t :: \langle 'a \text{ term} \rangle$ **and** $ts :: \langle 'a \text{ term list} \rangle$
shows $\langle \text{psubstt } id \ t = t \rangle$ **and** $\langle \text{psubstts } (\lambda x. x) \ ts = ts \rangle$
 $\langle \text{proof} \rangle$

theorem *psubst-id* [*simp*]: $\langle \text{psubst } id = id \rangle$

$\langle \text{proof} \rangle$

theorem *psubstt-image* [*simp*]:

$\langle \text{paramst } (\text{psubstt } f \ t) = f \text{ ' paramst } t \rangle$
 $\langle \text{paramsts } (\text{psubstts } f \ ts) = f \text{ ' paramsts } ts \rangle$
 $\langle \text{proof} \rangle$

theorem *psubst-image* [*simp*]: $\langle \text{params } (\text{psubst } f \ p) = f \text{ ' params } p \rangle$

$\langle \text{proof} \rangle$

4 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 $shift :: \langle (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a \rangle (-\langle -: \rangle) [90, 0, 0] 91$
where

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

lemma $shift\text{-}eq [simp]: \langle i = j \implies (e\langle i:T \rangle) j = T \rangle$
 $\langle proof \rangle$

lemma $shift\text{-}gt [simp]: \langle j < i \implies (e\langle i:T \rangle) j = e \text{ } j \rangle$
 $\langle proof \rangle$

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

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

primrec

$evalt :: \langle (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \text{ list} \Rightarrow 'c) \Rightarrow 'a \text{ term} \Rightarrow 'c \rangle$ **and**
 $evalts :: \langle (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \text{ list} \Rightarrow 'c) \Rightarrow 'a \text{ term list} \Rightarrow 'c \text{ list} \rangle$ **where**
 $\langle evalt \text{ } e \text{ } f \text{ } (Var \text{ } n) = e \text{ } n \rangle$
 $| \langle evalt \text{ } e \text{ } f \text{ } (App \text{ } a \text{ } ts) = f \text{ } a \text{ } (evalts \text{ } e \text{ } f \text{ } ts) \rangle$
 $| \langle evalts \text{ } e \text{ } f \text{ } [] = [] \rangle$
 $| \langle evalts \text{ } e \text{ } f \text{ } (t \text{ } \# \text{ } ts) = evalt \text{ } e \text{ } f \text{ } t \text{ } \# \text{ } evalts \text{ } e \text{ } f \text{ } ts \rangle$

primrec $eval :: \langle (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \text{ list} \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c \text{ list} \Rightarrow bool) \Rightarrow ('a, 'b) \text{ form} \Rightarrow bool \rangle$ **where**

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

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

definition $model :: \langle (nat \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \text{ list} \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c \text{ list} \Rightarrow bool) \Rightarrow$

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

The following substitution lemmas relate substitution and evaluation functions:

theorem *subst-lemma'* [simp]:

$\langle \text{evalt } e \text{ f } (\text{substt } t \text{ u } i) = \text{evalt } (e \langle i: \text{evalt } e \text{ f } u \rangle) \text{ f } t \rangle$
 $\langle \text{evalts } e \text{ f } (\text{substts } ts \text{ u } i) = \text{evalts } (e \langle i: \text{evalt } e \text{ f } u \rangle) \text{ f } ts \rangle$
 $\langle \text{proof} \rangle$

theorem *lift-lemma* [simp]:

$\langle \text{evalt } (e \langle 0:z \rangle) \text{ f } (\text{liftt } t) = \text{evalt } e \text{ f } t \rangle$
 $\langle \text{evalts } (e \langle 0:z \rangle) \text{ f } (\text{liftts } ts) = \text{evalts } e \text{ f } ts \rangle$
 $\langle \text{proof} \rangle$

theorem *subst-lemma* [simp]:

$\langle \text{eval } e \text{ f } g (\text{subst } a \text{ t } i) = \text{eval } (e \langle i: \text{evalt } e \text{ f } t \rangle) \text{ f } g \text{ a} \rangle$
 $\langle \text{proof} \rangle$

theorem *upd-lemma'* [simp]:

$\langle n \notin \text{paramst } t \Longrightarrow \text{evalt } e (f(n := x)) \text{ t} = \text{evalt } e \text{ f } t \rangle$
 $\langle n \notin \text{paramsts } ts \Longrightarrow \text{evalts } e (f(n := x)) \text{ ts} = \text{evalts } e \text{ f } ts \rangle$
 $\langle \text{proof} \rangle$

theorem *upd-lemma* [simp]:

$\langle n \notin \text{params } p \Longrightarrow \text{eval } e (f(n := x)) \text{ g } p = \text{eval } e \text{ f } g \text{ p} \rangle$
 $\langle \text{proof} \rangle$

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

$\text{list-all} (\text{eval } e (f(n := x)) \text{ g}) \text{ G} = \text{list-all} (\text{eval } e \text{ f } g) \text{ G} \rangle$
 $\langle \text{proof} \rangle$

theorem *psubst-eval'* [simp]:

$\langle \text{evalt } e \text{ f } (\text{psubstt } h \text{ t}) = \text{evalt } e (\lambda p. f (h \text{ p})) \text{ t} \rangle$
 $\langle \text{evalts } e \text{ f } (\text{psubstts } h \text{ ts}) = \text{evalts } e (\lambda p. f (h \text{ p})) \text{ ts} \rangle$
 $\langle \text{proof} \rangle$

theorem *psubst-eval*:

$\langle \text{eval } e \text{ f } g (\text{psubst } h \text{ p}) = \text{eval } e (\lambda p. f (h \text{ p})) \text{ g } p \rangle$
 $\langle \text{proof} \rangle$

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

theorem *ex-all-commute-eval*:

$\langle \text{eval } e \text{ f } g (\text{Impl} (\text{Exists} (\text{Forall} (\text{Pred } p [\text{Var } 1, \text{Var } 0])))$
 $(\text{Forall} (\text{Exists} (\text{Pred } p [\text{Var } 0, \text{Var } 1]))) \rangle$
 $\langle \text{proof} \rangle$

5 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* :: $\langle ('a, 'b) \text{ form list} \Rightarrow ('a, 'b) \text{ form} \Rightarrow \text{bool} \rangle$ ($- \vdash -$ [50,50] 50)
where

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

The following derived inference rules are sometimes useful in applications.

theorem *Class'*: $\langle \text{Neg } A \# G \vdash A \Longrightarrow G \vdash A \rangle$
 $\langle \text{proof} \rangle$

theorem *cut*: $\langle G \vdash A \Longrightarrow A \# G \vdash B \Longrightarrow G \vdash B \rangle$
 $\langle \text{proof} \rangle$

theorem *ForallE'*: $\langle G \vdash \text{Forall } a \Longrightarrow \text{subst } a \ t \ 0 \# G \vdash B \Longrightarrow G \vdash B \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle [] \vdash \text{Or } (\text{Pred } p \ []) \ (\text{Neg } (\text{Pred } p \ [])) \rangle$ (**is** $\langle - \vdash ?\text{or} \rangle$)
 $\langle \text{proof} \rangle$

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

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

theorem peirce:
 $\langle [] \vdash \text{Impl (Impl (Impl (Pred P []) (Pred Q [])) (Pred P [])) (Pred P [])} \rangle$
 $\langle \text{is } \langle [] \vdash \text{Impl ?PQP (Pred P [])} \rangle \rangle$
 $\langle \text{proof} \rangle$

6 Correctness

The correctness of the proof calculus introduced in §5 can now be proved by induction on the derivation of $G \vdash p$, using the substitution rules proved in §4.

theorem correctness: $\langle G \vdash p \implies \forall e f g. e, f, g, G \models p \rangle$
 $\langle \text{proof} \rangle$

7 Completeness

The goal of this section is to prove completeness of the natural deduction calculus introduced in §5. 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 $\text{Neg } p, ps \not\vdash FF$ implies that $\text{Neg } p, ps$ is consistent, which, by the model existence theorem, implies that $\text{Neg } p, ps$ has a model, which in turn implies that $ps \not\models p$. By contraposition, it therefore follows from $ps \models p$ that $\text{Neg } p, ps \vdash FF$, 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 §7.1).

The key idea behind the proof of the model existence theorem is to extend a consistent set to one that is *maximal* (see §7.5). In order to do this, we use the fact that the set of formulae is enumerable (see §7.4), 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 §7.2) and finiteness (see §7.3) properties are needed. It can be shown that a maximal consistent set is a *Hintikka set* (see §7.6). Hintikka sets are satisfiable in *Herbrand* models, where closed terms coincide with their interpretation.

7.1 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* :: $\langle ('a, 'b) \text{ form set set} \Rightarrow \text{bool} \rangle$ **where**

$$\langle \text{consistency } C = (\forall S. S \in C \longrightarrow$$

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

In §7.3, 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* :: $\langle ('a, 'b) \text{ form set set} \Rightarrow \text{bool} \rangle$ **where**

$$\langle \text{alt-consistency } C = (\forall S. S \in C \longrightarrow$$

$$\begin{aligned} & (\forall p \text{ ts. } \neg (\text{Pred } p \text{ ts} \in S \wedge \text{Neg } (\text{Pred } p \text{ ts}) \in S)) \wedge \\ & FF \notin S \wedge \text{Neg } TT \notin S \wedge \\ & (\forall Z. \text{Neg } (\text{Neg } Z) \in S \longrightarrow S \cup \{Z\} \in C) \wedge \\ & (\forall A B. \text{And } A B \in S \longrightarrow S \cup \{A, B\} \in C) \wedge \\ & (\forall A B. \text{Neg } (\text{Or } A B) \in S \longrightarrow S \cup \{\text{Neg } A, \text{Neg } B\} \in C) \wedge \\ & (\forall A B. \text{Or } A B \in S \longrightarrow S \cup \{A\} \in C \vee S \cup \{B\} \in C) \wedge \\ & (\forall A B. \text{Neg } (\text{And } A B) \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{\text{Neg } B\} \in C) \wedge \\ & (\forall A B. \text{Impl } A B \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{B\} \in C) \wedge \end{aligned}$$

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

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* :: $\langle 'a, 'b \text{ form set set} \Rightarrow 'a, 'b \text{ form set set} \rangle$
where

$\langle \text{mk-alt-consistency } C = \{S. \exists f. \text{psubst } f ' S \in C\} \rangle$

theorem *alt-consistency*:

assumes *conc*: $\langle \text{consistency } C \rangle$

shows $\langle \text{alt-consistency} (\text{mk-alt-consistency } C) \rangle$ (**is** $\langle \text{alt-consistency } ?C' \rangle$)

$\langle \text{proof} \rangle$

theorem *mk-alt-consistency-subset*: $\langle C \subseteq \text{mk-alt-consistency } C \rangle$

$\langle \text{proof} \rangle$

7.2 Closure under subsets

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

definition *close* :: $\langle 'a, 'b \text{ form set set} \Rightarrow 'a, 'b \text{ form set set} \rangle$ **where**

$\langle \text{close } C = \{S. \exists S' \in C. S \subseteq S'\} \rangle$

definition *subset-closed* :: $\langle 'a \text{ set set} \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{subset-closed } C = (\forall S' \in C. \forall S. S \subseteq S' \longrightarrow S \in C) \rangle$

lemma *subset-in-close*:

assumes $\langle S \subseteq S' \rangle$

shows $\langle S' \cup x \in C \longrightarrow S \cup x \in \text{close } C \rangle$

$\langle \text{proof} \rangle$

theorem *close-consistency*:

assumes *conc*: $\langle \text{consistency } C \rangle$

shows $\langle \text{consistency} (\text{close } C) \rangle$

$\langle \text{proof} \rangle$

theorem *close-closed*: $\langle \text{subset-closed} (\text{close } C) \rangle$

$\langle \text{proof} \rangle$

theorem *close-subset*: $\langle C \subseteq \text{close } C \rangle$
 $\langle \text{proof} \rangle$

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

theorem *mk-alt-consistency-closed*:
assumes $\langle \text{subset-closed } C \rangle$
shows $\langle \text{subset-closed } (\text{mk-alt-consistency } C) \rangle$
 $\langle \text{proof} \rangle$

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

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

theorem *finite-alt-consistency*:
assumes *altconc*: $\langle \text{alt-consistency } C \rangle$
and $\langle \text{subset-closed } C \rangle$
shows $\langle \text{alt-consistency } (\text{mk-finite-char } C) \rangle$
 $\langle \text{proof} \rangle$

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

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

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

7.4 Enumerating datatypes

As has already been mentioned earlier, the proof of the model existence theorem relies on the fact that the set of formulae is enumerable. Using the infrastructure for datatypes, the types *FOL-Fitting.term* and *form* can automatically be shown to be a member of the *countable* type class:

instance $\langle \text{term} \rangle$:: $(\text{countable}) \text{ countable}$
 $\langle \text{proof} \rangle$

instance *form* :: $(\text{countable}, \text{countable}) \text{ countable}$

⟨proof⟩

7.5 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* $f \implies x \in f\ m \implies x \in f\ (m + n)$ ⟩
⟨proof⟩

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

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

lemma *chain-union-closed'*:
assumes ⟨*is-chain* f ⟩ **and** ⟨ $(\forall n. f\ n \in C)$ ⟩ **and** ⟨ $\forall S' \in C. \forall S \subseteq S'. S \in C$ ⟩
and ⟨finite S' ⟩ **and** ⟨ $S' \subseteq (\bigcup n. f\ n)$ ⟩
shows ⟨ $S' \in C$ ⟩
⟨proof⟩

theorem *chain-union-closed*:
assumes ⟨finite-char C ⟩ **and** ⟨*is-chain* f ⟩ **and** ⟨ $\forall n. f\ n \in C$ ⟩
shows ⟨ $(\bigcup n. f\ n) \in C$ ⟩
⟨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 (*nonexhaustive*) *dest-Neg* :: ⟨('a, 'b) form ⇒ ('a, 'b) form⟩ **where**
⟨*dest-Neg* (*Neg* p) = p ⟩

primrec (*nonexhaustive*) *dest-Forall* :: ⟨('a, 'b) form ⇒ ('a, 'b) form⟩ **where**
⟨*dest-Forall* (*Forall* p) = p ⟩

primrec (*nonexhaustive*) *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* $S\ C\ f\ 0 = S$ ⟩

| $\langle \text{extend } S C f (Suc n) = (\text{if } \text{extend } S C f n \cup \{f n\} \in C$
 then
 (if $(\exists p. f n = \text{Exists } p)$
 then $\text{extend } S C f n \cup \{f n\} \cup \{\text{subst } (\text{dest-Exists } (f n))$
 (App (SOME $k. k \notin (\bigcup p \in \text{extend } S C f n \cup \{f n\}. \text{params } p)) \ [] \ 0)$
 else if $(\exists p. f n = \text{Neg } (\text{Forall } p))$
 then $\text{extend } S C f n \cup \{f n\} \cup \{\text{Neg } (\text{subst } (\text{dest-Forall } (\text{dest-Neg } (f n)))$
 (App (SOME $k. k \notin (\bigcup p \in \text{extend } S C f n \cup \{f n\}. \text{params } p)) \ [] \ 0)$
 else $\text{extend } S C f n \cup \{f n\}$
 else $\text{extend } S C f n \rangle$

definition *Extend* :: $\langle (\text{nat}, 'b) \text{ form set} \Rightarrow (\text{nat}, 'b) \text{ form set set} \Rightarrow$
 $(\text{nat} \Rightarrow (\text{nat}, 'b) \text{ form}) \Rightarrow (\text{nat}, 'b) \text{ form set} \rangle$ **where**
 $\langle \text{Extend } S C f = (\bigcup n. \text{extend } S C f n) \rangle$

theorem *is-chain-extend*: $\langle \text{is-chain } (\text{extend } S C f) \rangle$
 $\langle \text{proof} \rangle$

theorem *finite-paramst [simp]*: $\langle \text{finite } (\text{paramst } (t :: 'a \text{ term})) \rangle$
 $\langle \text{finite } (\text{paramsts } (ts :: 'a \text{ term list})) \rangle$
 $\langle \text{proof} \rangle$

theorem *finite-params [simp]*: $\langle \text{finite } (\text{params } p) \rangle$
 $\langle \text{proof} \rangle$

theorem *finite-params-extend [simp]*:
 $\langle \text{infinite } (\bigcap p \in S. \text{params } p) \Longrightarrow \text{infinite } (\bigcap p \in \text{extend } S C f n. \text{params } p) \rangle$
 $\langle \text{proof} \rangle$

lemma *infinite-params-available*:
assumes $\langle \text{infinite } (\text{params } p) \rangle$
shows $\langle \exists x. x \notin (\bigcup p \in \text{extend } S C f n \cup \{f n\}. \text{params } p) \rangle$
 $\langle \text{proof} \rangle$

lemma *extend-in-C-Exists*:
assumes $\langle \text{alt-consistency } C \rangle$
and $\langle \text{infinite } (\text{params } p) \rangle$
and $\langle \text{extend } S C f n \cup \{f n\} \in C \rangle$ (**is** $\langle ?S' \in C \rangle$)
and $\langle \exists p. f n = \text{Exists } p \rangle$
shows $\langle \text{extend } S C f (Suc n) \in C \rangle$
 $\langle \text{proof} \rangle$

lemma *extend-in-C-Neg-Forall*:
assumes $\langle \text{alt-consistency } C \rangle$
and $\langle \text{infinite } (\text{params } p) \rangle$
and $\langle \text{extend } S C f n \cup \{f n\} \in C \rangle$ (**is** $\langle ?S' \in C \rangle$)
and $\langle \forall p. f n \neq \text{Exists } p \rangle$
and $\langle \exists p. f n = \text{Neg } (\text{Forall } p) \rangle$
shows $\langle \text{extend } S C f (Suc n) \in C \rangle$

$\langle \text{proof} \rangle$

lemma *extend-in-C-no-delta*:

assumes $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \in C \rangle$

and $\langle \forall p. f \ n \neq \text{Exists } p \rangle$

and $\langle \forall p. f \ n \neq \text{Neg } (\text{Forall } p) \rangle$

shows $\langle \text{extend } S \ C \ f \ (\text{Suc } n) \in C \rangle$

$\langle \text{proof} \rangle$

lemma *extend-in-C-stop*:

assumes $\langle \text{extend } S \ C \ f \ n \in C \rangle$

and $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \notin C \rangle$

shows $\langle \text{extend } S \ C \ f \ (\text{Suc } n) \in C \rangle$

$\langle \text{proof} \rangle$

theorem *extend-in-C*: $\langle \text{alt-consistency } C \implies$

$S \in C \implies \text{infinite } (- (\bigcup p \in S. \text{params } p)) \implies \text{extend } S \ C \ f \ n \in C \rangle$

$\langle \text{proof} \rangle$

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*: $\langle \text{alt-consistency } C \implies \text{finite-char } C \implies$

$S \in C \implies \text{infinite } (- (\bigcup p \in S. \text{params } p)) \implies \text{Extend } S \ C \ f \in C \rangle$

$\langle \text{proof} \rangle$

theorem *Extend-subset*: $\langle S \subseteq \text{Extend } S \ C \ f \rangle$

$\langle \text{proof} \rangle$

The *Extend* function yields a maximal set:

definition *maximal* :: $\langle 'a \ \text{set} \Rightarrow 'a \ \text{set} \ \text{set} \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{maximal } S \ C = (\forall S' \in C. S \subseteq S' \longrightarrow S = S') \rangle$

theorem *extend-maximal*:

assumes $\langle \forall y. \exists n. y = f \ n \rangle$

and $\langle \text{finite-char } C \rangle$

shows $\langle \text{maximal } (\text{Extend } S \ C \ f) \ C \rangle$

$\langle \text{proof} \rangle$

7.6 Hintikka sets and Herbrand models

A Hintikka set is defined as follows:

definition *hintikka* :: $\langle ('a, 'b) \ \text{form} \ \text{set} \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{hintikka } H =$

$((\forall p \ ts. \neg (\text{Pred } p \ ts \in H \wedge \text{Neg } (\text{Pred } p \ ts) \in H)) \wedge$

$FF \notin H \wedge \text{Neg } TT \notin H \wedge$

$(\forall Z. \text{Neg } (\text{Neg } Z) \in H \longrightarrow Z \in H) \wedge$

$(\forall A \ B. \text{And } A \ B \in H \longrightarrow A \in H \wedge B \in H) \wedge$

$$\begin{aligned}
& (\forall A B. \text{Neg} (\text{Or } A B) \in H \longrightarrow \text{Neg } A \in H \wedge \text{Neg } B \in H) \wedge \\
& (\forall A B. \text{Or } A B \in H \longrightarrow A \in H \vee B \in H) \wedge \\
& (\forall A B. \text{Neg} (\text{And } A B) \in H \longrightarrow \text{Neg } A \in H \vee \text{Neg } B \in H) \wedge \\
& (\forall A B. \text{Impl } A B \in H \longrightarrow \text{Neg } A \in H \vee B \in H) \wedge \\
& (\forall A B. \text{Neg} (\text{Impl } A B) \in H \longrightarrow A \in H \wedge \text{Neg } B \in H) \wedge \\
& (\forall P t. \text{closedt } 0 t \longrightarrow \text{Forall } P \in H \longrightarrow \text{subst } P t 0 \in H) \wedge \\
& (\forall P t. \text{closedt } 0 t \longrightarrow \text{Neg} (\text{Exists } P) \in H \longrightarrow \text{Neg} (\text{subst } P t 0) \in H) \wedge \\
& (\forall P. \text{Exists } P \in H \longrightarrow (\exists t. \text{closedt } 0 t \wedge \text{subst } P t 0 \in H)) \wedge \\
& (\forall P. \text{Neg} (\text{Forall } P) \in H \longrightarrow (\exists t. \text{closedt } 0 t \wedge \text{Neg} (\text{subst } P t 0) \in H))
\end{aligned}$$

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 §3, 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 ⇒ 'a term>* **and**
terms-of-hterms :: *<'a hterm list ⇒ '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 ⇒ 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>

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*:

assumes *hin*: *<hintikka H>*
shows *<(p ∈ H ⇒ closed 0 p ⇒ eval e HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H) p) ∧*

$(Neg\ p \in H \longrightarrow closed\ 0\ p \longrightarrow$
 $eval\ e\ HApp\ (\lambda a\ ts.\ Pred\ a\ (terms-of-hterms\ ts) \in H)\ (Neg\ p))\rangle$
 $\langle proof \rangle$

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

lemma *Exists-in-extend*:

assumes $\langle extend\ S\ C\ f\ n\ \cup\ \{f\ n\} \in C \rangle$ (**is** $\langle ?S' \in C \rangle$)
and $\langle Exists\ P = f\ n \rangle$
shows $\langle P[(App\ (SOME\ k.\ k \notin (\bigcup p \in extend\ S\ C\ f\ n\ \cup\ \{f\ n\}.\ params\ p))\ [])/0]$
 \in
 $extend\ S\ C\ f\ (Suc\ n)\rangle$
(is $\langle subst\ P\ ?t\ 0 \in extend\ S\ C\ f\ (Suc\ n)\rangle$)
 $\langle proof \rangle$

lemma *Neg-Forall-in-extend*:

assumes $\langle extend\ S\ C\ f\ n\ \cup\ \{f\ n\} \in C \rangle$ (**is** $\langle ?S' \in C \rangle$)
and $\langle Neg\ (Forall\ P) = f\ n \rangle$
shows $\langle Neg\ (P[(App\ (SOME\ k.\ k \notin (\bigcup p \in extend\ S\ C\ f\ n\ \cup\ \{f\ n\}.\ params\ p))\ [])/0]) \in$
 $extend\ S\ C\ f\ (Suc\ n)\rangle$
(is $\langle Neg\ (subst\ P\ ?t\ 0) \in extend\ S\ C\ f\ (Suc\ n)\rangle$)
 $\langle proof \rangle$

theorem *extend-hintikka*:

assumes *fin-ch*: $\langle finite-char\ C \rangle$
and *infin-p*: $\langle infinite\ (\neg (\bigcup p \in S.\ params\ p)) \rangle$
and *surj*: $\langle \forall y.\ \exists n.\ y = f\ n \rangle$
and *altc*: $\langle alt-consistency\ C \rangle$
and $\langle S \in C \rangle$
shows $\langle hintikka\ (Extend\ S\ C\ f) \rangle$ (**is** $\langle hintikka\ ?H \rangle$)
 $\langle proof \rangle$

7.7 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:

lemma *hintikka-Extend-S*:

assumes $\langle consistency\ C \rangle$ **and** $\langle S \in C \rangle$
and $\langle infinite\ (\neg (\bigcup p \in S.\ params\ p)) \rangle$
shows $\langle hintikka\ (Extend\ S\ (mk-finite-char\ (mk-alt-consistency\ (close\ C)))\ from-nat) \rangle$
(is $\langle hintikka\ (Extend\ S\ ?C'\ from-nat) \rangle$)
 $\langle proof \rangle$

theorem *model-existence*:

assumes $\langle consistency\ C \rangle$
and $\langle S \in C \rangle$
and $\langle infinite\ (\neg (\bigcup p \in S.\ params\ p)) \rangle$

and $\langle p \in S \rangle$
and $\langle \text{closed } 0 \ p \rangle$
shows $\langle \text{eval } e \text{ HApp } (\lambda a \ ts. \text{Pred } a \ (\text{terms-of-hterms } ts)) \in \text{Extend } S$
 $(\text{mk-finite-char } (\text{mk-alt-consistency } (\text{close } C))) \text{ from-nat } p \rangle$
 $\langle \text{proof} \rangle$

7.8 Completeness for Natural Deduction

Thanks to the model existence theorem, we can now show the completeness of the natural deduction calculus introduced in §5. 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*: $\langle \text{infinite } (\text{UNIV} :: 'a \ \text{set}) \rangle$
shows $\langle \text{consistency } \{S :: ('a, 'b) \ \text{form set. } \exists G. S = \text{set } G \wedge \neg G \vdash FF\} \rangle$
 $\langle \text{proof} \rangle$

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

theorem *natded-complete*:
assumes $\langle \text{closed } 0 \ p \rangle$
and $\langle \text{list-all } (\text{closed } 0) \ ps \rangle$
and *mod*: $\langle \forall e \ f \ g. e, (f :: \text{nat} \Rightarrow \text{nat hterm list} \Rightarrow \text{nat hterm}),$
 $(g :: \text{nat} \Rightarrow \text{nat hterm list} \Rightarrow \text{bool}), ps \models p \rangle$
shows $\langle ps \vdash p \rangle$
 $\langle \text{proof} \rangle$

8 Löwenheim-Skolem theorem

Another application of the model existence theorem presented in §7.7 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*:
 $\langle \text{consistency } \{S. \text{infinite } (\neg (\bigcup p \in S. \text{params } p)) \wedge (\exists f. \forall (p :: ('a, 'b) \ \text{form}) \in S. \text{eval } e \ f \ g \ p)\} \rangle$
 $\langle \text{proof} \rangle$

theorem *doublep-infinite-params*:
 $\langle \text{infinite } (\neg (\bigcup p \in \text{psubst } (\lambda n :: \text{nat}. 2 * n) ' S. \text{params } p)) \rangle$
 $\langle \text{proof} \rangle$

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.

theorem *loewenheim-skolem*:

assumes *evalS*: $\langle \forall p \in S. \text{eval } e \ f \ g \ p \rangle$
shows $\langle \forall p \in S. \text{closed } 0 \ p \longrightarrow \text{eval } e' \ (\lambda n. \text{HApp } (2 * n)) \ (\lambda a \ ts. \text{Pred } a \ (\text{terms-of-hterms } ts) \in \text{Extend } (\text{psubst } (\lambda n. 2 * n) \ 'S) \ (\text{mk-finite-char } (\text{mk-alt-consistency } (\text{close } \{S. \text{infinite } (- \ (\bigcup p \in S. \text{params } p)) \wedge (\exists f. \forall p \in S. \text{eval } e \ f \ g \ p)\}))) \text{from-nat}) \ p \rangle$
(is $\langle \forall - \in -. \text{---} \longrightarrow \text{eval } - \ ?g \ - \rangle$)
proof

9 Completeness for open formulas

abbreviation $\langle \text{new-term } c \ t \equiv c \notin \text{paramst } t \rangle$

abbreviation $\langle \text{new-list } c \ ts \equiv c \notin \text{paramsts } ts \rangle$

abbreviation $\langle \text{new } c \ p \equiv c \notin \text{params } p \rangle$

abbreviation $\langle \text{news } c \ z \equiv \text{list-all } (\text{new } c) \ z \rangle$

9.1 Renaming

lemma *new-psubst-image'*:

$\langle \text{new-term } c \ t \implies d \notin \text{image } f \ (\text{paramst } t) \implies \text{new-term } d \ (\text{psubstt } (f(c := d)) \ t) \rangle$
 $\langle \text{new-list } c \ l \implies d \notin \text{image } f \ (\text{paramsts } l) \implies \text{new-list } d \ (\text{psubstts } (f(c := d)) \ l) \rangle$
proof

lemma *new-psubst-image*: $\langle \text{new } c \ p \implies d \notin \text{image } f \ (\text{params } p) \implies \text{new } d \ (\text{psubst } (f(c := d)) \ p) \rangle$
proof

lemma *news-psubst*: $\langle \text{news } c \ z \implies d \notin \text{image } f \ (\bigcup p \in \text{set } z. \text{params } p) \implies \text{news } d \ (\text{map } (\text{psubst } (f(c := d))) \ z) \rangle$
proof

lemma *member-psubst*: $\langle p \in \text{set } z \implies \text{psubst } f \ p \in \text{set } (\text{map } (\text{psubst } f) \ z) \rangle$
proof

lemma *deriv-psubst*:

fixes $f :: \langle 'a \Rightarrow 'a \rangle$
assumes *inf-params*: $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$
shows $\langle z \vdash p \implies \text{map } (\text{psubst } f) \ z \vdash \text{psubst } f \ p \rangle$
proof

9.2 Substitution for constants

primrec

subc-term :: $\langle 'a \Rightarrow 'a \text{ term} \Rightarrow 'a \text{ term} \Rightarrow 'a \text{ term} \rangle$ **and**

subc-list :: $\langle 'a \Rightarrow 'a \text{ term} \Rightarrow 'a \text{ term list} \Rightarrow 'a \text{ term list} \rangle$ **where**
 $\langle \text{subc-term } c \ s \ (\text{Var } n) = \text{Var } n \rangle \mid$
 $\langle \text{subc-term } c \ s \ (\text{App } i \ l) = (\text{if } i = c \ \text{then } s \ \text{else } \text{App } i \ (\text{subc-list } c \ s \ l)) \rangle \mid$
 $\langle \text{subc-list } c \ s \ [] = [] \rangle \mid$
 $\langle \text{subc-list } c \ s \ (t \ \# \ l) = \text{subc-term } c \ s \ t \ \# \ \text{subc-list } c \ s \ l \rangle$

primrec *subc* :: $\langle 'a \Rightarrow 'a \text{ term} \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$ **where**
 $\langle \text{subc } c \ s \ FF = FF \rangle \mid$
 $\langle \text{subc } c \ s \ TT = TT \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Pred } i \ l) = \text{Pred } i \ (\text{subc-list } c \ s \ l) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Neg } p) = \text{Neg } (\text{subc } c \ s \ p) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Impl } p \ q) = \text{Impl } (\text{subc } c \ s \ p) \ (\text{subc } c \ s \ q) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Or } p \ q) = \text{Or } (\text{subc } c \ s \ p) \ (\text{subc } c \ s \ q) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{And } p \ q) = \text{And } (\text{subc } c \ s \ p) \ (\text{subc } c \ s \ q) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Exists } p) = \text{Exists } (\text{subc } c \ (\text{liftt } s) \ p) \rangle \mid$
 $\langle \text{subc } c \ s \ (\text{Forall } p) = \text{Forall } (\text{subc } c \ (\text{liftt } s) \ p) \rangle$

primrec *subcs* :: $\langle 'a \Rightarrow 'a \text{ term} \Rightarrow ('a, 'b) \text{ form list} \Rightarrow ('a, 'b) \text{ form list} \rangle$ **where**
 $\langle \text{subcs } c \ s \ [] = [] \rangle \mid$
 $\langle \text{subcs } c \ s \ (p \ \# \ z) = \text{subc } c \ s \ p \ \# \ \text{subcs } c \ s \ z \rangle$

lemma *subst-0-lift*:
 $\langle \text{subst } (\text{liftt } t) \ s \ 0 = t \rangle$
 $\langle \text{substts } (\text{liftts } l) \ s \ 0 = l \rangle$
 $\langle \text{proof} \rangle$

lemma *params-lift* [*simp*]:
fixes $t :: \langle 'a \text{ term} \rangle$ **and** $ts :: \langle 'a \text{ term list} \rangle$
shows
 $\langle \text{paramst } (\text{liftt } t) = \text{paramst } t \rangle$
 $\langle \text{paramsts } (\text{liftts } ts) = \text{paramsts } ts \rangle$
 $\langle \text{proof} \rangle$

lemma *subst-new'* [*simp*]:
 $\langle \text{new-term } c \ s \Longrightarrow \text{new-term } c \ t \Longrightarrow \text{new-term } c \ (\text{subst } t \ s \ m) \rangle$
 $\langle \text{new-term } c \ s \Longrightarrow \text{new-list } c \ l \Longrightarrow \text{new-list } c \ (\text{substts } l \ s \ m) \rangle$
 $\langle \text{proof} \rangle$

lemma *subst-new* [*simp*]: $\langle \text{new-term } c \ s \Longrightarrow \text{new } c \ p \Longrightarrow \text{new } c \ (\text{subst } p \ s \ m) \rangle$
 $\langle \text{proof} \rangle$

lemma *subst-new-all*:
assumes $\langle a \notin \text{set } cs \rangle$ $\langle \text{list-all } (\lambda c. \text{new } c \ p) \ cs \rangle$
shows $\langle \text{list-all } (\lambda c. \text{new } c \ (\text{subst } p \ (\text{App } a \ []) \ m)) \ cs \rangle$
 $\langle \text{proof} \rangle$

lemma *subc-new'* [*simp*]:
 $\langle \text{new-term } c \ t \Longrightarrow \text{subc-term } c \ s \ t = t \rangle$
 $\langle \text{new-list } c \ l \Longrightarrow \text{subc-list } c \ s \ l = l \rangle$

$\langle \text{proof} \rangle$

lemma *subc-new* [*simp*]: $\langle \text{new } c \ p \implies \text{subc } c \ s \ p = p \rangle$
 $\langle \text{proof} \rangle$

lemma *subcs-news*: $\langle \text{news } c \ z \implies \text{subcs } c \ s \ z = z \rangle$
 $\langle \text{proof} \rangle$

lemma *subc-psubst'* [*simp*]:
 $\langle (\forall x \in \text{paramst } t. x \neq c \longrightarrow f \ x \neq f \ c) \implies$
 $\text{psubstt } f \ (\text{subc-term } c \ s \ t) = \text{subc-term } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubstt } f \ t) \rangle$
 $\langle (\forall x \in \text{paramsts } l. x \neq c \longrightarrow f \ x \neq f \ c) \implies$
 $\text{psubstts } f \ (\text{subc-list } c \ s \ l) = \text{subc-list } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubstts } f \ l) \rangle$
 $\langle \text{proof} \rangle$

lemma *subc-psubst*: $\langle (\forall x \in \text{params } p. x \neq c \longrightarrow f \ x \neq f \ c) \implies$
 $\text{psubst } f \ (\text{subc } c \ s \ p) = \text{subc } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubst } f \ p) \rangle$
 $\langle \text{proof} \rangle$

lemma *subcs-psubst*: $\langle (\forall x \in (\bigcup p \in \text{set } z. \text{params } p). x \neq c \longrightarrow f \ x \neq f \ c) \implies$
 $\text{map } (\text{psubst } f) \ (\text{subcs } c \ s \ z) = \text{subcs } (f \ c) \ (\text{psubstt } f \ s) \ (\text{map } (\text{psubst } f) \ z) \rangle$
 $\langle \text{proof} \rangle$

lemma *new-lift*:
 $\langle \text{new-term } c \ t \implies \text{new-term } c \ (\text{liftt } t) \rangle$
 $\langle \text{new-list } c \ l \implies \text{new-list } c \ (\text{liftts } l) \rangle$
 $\langle \text{proof} \rangle$

lemma *new-subc'* [*simp*]:
 $\langle \text{new-term } d \ s \implies \text{new-term } d \ t \implies \text{new-term } d \ (\text{subc-term } c \ s \ t) \rangle$
 $\langle \text{new-term } d \ s \implies \text{new-list } d \ l \implies \text{new-list } d \ (\text{subc-list } c \ s \ l) \rangle$
 $\langle \text{proof} \rangle$

lemma *new-subc* [*simp*]: $\langle \text{new-term } d \ s \implies \text{new } d \ p \implies \text{new } d \ (\text{subc } c \ s \ p) \rangle$
 $\langle \text{proof} \rangle$

lemma *news-subcs*: $\langle \text{new-term } d \ s \implies \text{news } d \ z \implies \text{news } d \ (\text{subcs } c \ s \ z) \rangle$
 $\langle \text{proof} \rangle$

lemma *psubst-new-free'*:
 $\langle c \neq n \implies \text{new-term } n \ (\text{psubstt } (\text{id}(n := c)) \ t) \rangle$
 $\langle c \neq n \implies \text{new-list } n \ (\text{psubstts } (\text{id}(n := c)) \ l) \rangle$
 $\langle \text{proof} \rangle$

lemma *psubst-new-free*: $\langle c \neq n \implies \text{new } n \ (\text{psubst } (\text{id}(n := c)) \ p) \rangle$
 $\langle \text{proof} \rangle$

lemma *map-psubst-new-free*: $\langle c \neq n \implies \text{news } n \ (\text{map } (\text{psubst } (\text{id}(n := c))) \ z) \rangle$
 $\langle \text{proof} \rangle$

lemma *psubst-new-away'* [simp]:

⟨new-term fresh t \implies psubstt (id(fresh := c)) (psubstt (id(c := fresh)) t) = t⟩
⟨new-list fresh l \implies psubstts (id(fresh := c)) (psubstts (id(c := fresh)) l) = l⟩
⟨proof⟩

lemma *psubst-new-away* [simp]: ⟨new fresh p \implies psubst (id(fresh := c)) (psubst (id(c := fresh)) p) = p⟩
⟨proof⟩

lemma *map-psubst-new-away*:

⟨news fresh z \implies map (psubst (id(fresh := c))) (map (psubst (id(c := fresh))) z) = z⟩
⟨proof⟩

lemma *psubst-new'*:

⟨new-term c t \implies psubstt (id(c := x)) t = t⟩
⟨new-list c l \implies psubstts (id(c := x)) l = l⟩
⟨proof⟩

lemma *psubst-new*: ⟨new c p \implies psubst (id(c := x)) p = p⟩
⟨proof⟩

lemma *map-psubst-new*: ⟨news c z \implies map (psubst (id(c := x))) z = z⟩
⟨proof⟩

lemma *lift-subst* [simp]:

⟨liftt (substt t u m) = substt (liftt t) (liftt u) (m + 1)⟩
⟨liftts (substts l u m) = substts (liftts l) (liftt u) (m + 1)⟩
⟨proof⟩

lemma *new-subc-same'* [simp]:

⟨new-term c s \implies new-term c (subc-term c s t)⟩
⟨new-term c s \implies new-list c (subc-list c s l)⟩
⟨proof⟩

lemma *new-subc-same*: ⟨new-term c s \implies new c (subc c s p)⟩
⟨proof⟩

lemma *lift-subc*:

⟨liftt (subc-term c s t) = subc-term c (liftt s) (liftt t)⟩
⟨liftts (subc-list c s l) = subc-list c (liftt s) (liftts l)⟩
⟨proof⟩

lemma *new-subc-put'*:

⟨new-term c s \implies subc-term c s (substt t u m) = subc-term c s (substt t (subc-term c s u) m)⟩
⟨new-term c s \implies subc-list c s (substts l u m) = subc-list c s (substts l (subc-term c s u) m)⟩

⟨proof⟩

lemma *new-subc-put*:

⟨new-term $c\ s \implies \text{subc } c\ s\ (\text{subst } p\ t\ m) = \text{subc } c\ s\ (\text{subst } p\ (\text{subc-term } c\ s\ t)\ m)$ ⟩

⟨proof⟩

lemma *subc-subst-new'*:

⟨new-term $c\ u \implies \text{subc-term } c\ (\text{substt } s\ u\ m)\ (\text{substt } t\ u\ m) = \text{substt } (\text{subc-term } c\ s\ t)\ u\ m$ ⟩

⟨new-term $c\ u \implies \text{subc-list } c\ (\text{substt } s\ u\ m)\ (\text{substts } l\ u\ m) = \text{substts } (\text{subc-list } c\ s\ l)\ u\ m$ ⟩

⟨proof⟩

lemma *subc-subst-new*:

⟨new-term $c\ t \implies \text{subc } c\ (\text{substt } s\ t\ m)\ (\text{subst } p\ t\ m) = \text{subst } (\text{subc } c\ s\ p)\ t\ m$ ⟩

⟨proof⟩

lemma *subc-sub-0-new* [simp]:

⟨new-term $c\ t \implies \text{subc } c\ s\ (\text{subst } p\ t\ 0) = \text{subst } (\text{subc } c\ (\text{liftt } s)\ p)\ t\ 0$ ⟩

⟨proof⟩

lemma *member-subc*: ⟨ $p \in \text{set } z \implies \text{subc } c\ s\ p \in \text{set } (\text{subcs } c\ s\ z)$ ⟩

⟨proof⟩

lemma *deriv-subc*:

fixes $p :: \langle ('a, 'b)\ \text{form} \rangle$

assumes *inf-params*: ⟨infinite (UNIV :: 'a set)⟩

shows ⟨ $z \vdash p \implies \text{subcs } c\ s\ z \vdash \text{subc } c\ s\ p$ ⟩

⟨proof⟩

9.3 Weakening assumptions

lemma *psubst-new-subset*:

assumes ⟨ $\text{set } z \subseteq \text{set } z' \ \langle c \notin (\bigcup p \in \text{set } z. \text{params } p) \rangle$ ⟩

shows ⟨ $\text{set } z \subseteq \text{set } (\text{map } (\text{psubst } (\text{id}(c := n)))\ z')$ ⟩

⟨proof⟩

lemma *subset-cons*: ⟨ $\text{set } z \subseteq \text{set } z' \implies \text{set } (p \# z) \subseteq \text{set } (p \# z')$ ⟩

⟨proof⟩

lemma *weaken-assumptions*:

fixes $p :: \langle ('a, 'b)\ \text{form} \rangle$

assumes *inf-params*: ⟨infinite (UNIV :: 'a set)⟩

shows ⟨ $z \vdash p \implies \text{set } z \subseteq \text{set } z' \implies z' \vdash p$ ⟩

⟨proof⟩

9.4 Implications and assumptions

primrec *put-imps* :: ⟨('a, 'b) form \Rightarrow ('a, 'b) form list \Rightarrow ('a, 'b) form⟩ **where**

$\langle \text{put-imps } p \ [] = p \rangle \mid$
 $\langle \text{put-imps } p \ (q \# z) = \text{Impl } q \ (\text{put-imps } p \ z) \rangle$

lemma *semantics-put-imps*:

$\langle (e, f, g, z \models p) = \text{eval } e \ f \ g \ (\text{put-imps } p \ z) \rangle$
 $\langle \text{proof} \rangle$

lemma *shift-imp-assum*:

fixes $p :: \langle 'a, 'b \rangle \text{ form} \rangle$
assumes *inf-params*: $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$
and $\langle z \vdash \text{Impl } p \ q \rangle$
shows $\langle p \# z \vdash q \rangle$
 $\langle \text{proof} \rangle$

lemma *remove-imps*:

assumes $\langle \text{infinite } (- \text{ params } p) \rangle$
shows $\langle z' \vdash \text{put-imps } p \ z \implies \text{rev } z \ @ \ z' \vdash p \rangle$
 $\langle \text{proof} \rangle$

9.5 Closure elimination

lemma *subc-sub-closed-var'* [simp]:

$\langle \text{new-term } c \ t \implies \text{closedt } (\text{Suc } m) \ t \implies \text{subc-term } c \ (\text{Var } m) \ (\text{subst } t \ (\text{App } c \ [])) \ m) = t \rangle$
 $\langle \text{new-list } c \ l \implies \text{closedts } (\text{Suc } m) \ l \implies \text{subc-list } c \ (\text{Var } m) \ (\text{substts } l \ (\text{App } c \ [])) \ m) = l \rangle$
 $\langle \text{proof} \rangle$

lemma *subc-sub-closed-var* [simp]: $\langle \text{new } c \ p \implies \text{closed } (\text{Suc } m) \ p \implies \text{subc } c \ (\text{Var } m) \ (\text{subst } p \ (\text{App } c \ [])) \ m) = p \rangle$
 $\langle \text{proof} \rangle$

primrec *put-unis* :: $\langle \text{nat} \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$ **where**

$\langle \text{put-unis } 0 \ p = p \rangle \mid$
 $\langle \text{put-unis } (\text{Suc } m) \ p = \text{Forall } (\text{put-unis } m \ p) \rangle$

lemma *sub-put-unis* [simp]:

$\langle \text{subst } (\text{put-unis } k \ p) \ (\text{App } c \ []) \ i = \text{put-unis } k \ (\text{subst } p \ (\text{App } c \ []) \ (i + k)) \rangle$
 $\langle \text{proof} \rangle$

lemma *closed-put-unis* [simp]: $\langle \text{closed } m \ (\text{put-unis } k \ p) = \text{closed } (m + k) \ p \rangle$
 $\langle \text{proof} \rangle$

lemma *valid-put-unis*: $\langle \forall (e :: \text{nat} \Rightarrow 'a) \ f \ g. \ \text{eval } e \ f \ g \ p \implies$

$\text{eval } (e :: \text{nat} \Rightarrow 'a) \ f \ g \ (\text{put-unis } m \ p) \rangle$
 $\langle \text{proof} \rangle$

lemma *put-unis-collapse*: $\langle \text{put-unis } m \ (\text{put-unis } n \ p) = \text{put-unis } (m + n) \ p \rangle$

$\langle \text{proof} \rangle$

fun *consts-for-unis* :: $\langle ('a, 'b) \text{ form} \Rightarrow 'a \text{ list} \Rightarrow ('a, 'b) \text{ form} \rangle$ **where**
 $\langle \text{consts-for-unis } (\text{Forall } p) (c \# cs) = \text{consts-for-unis } (\text{subst } p (\text{App } c []) 0) cs \rangle$ |
 $\langle \text{consts-for-unis } p - = p \rangle$

lemma *consts-for-unis*: $\langle [] \vdash \text{put-unis } (\text{length } cs) p \Longrightarrow$
 $[] \vdash \text{consts-for-unis } (\text{put-unis } (\text{length } cs) p) cs \rangle$
 $\langle \text{proof} \rangle$

primrec *vars-for-consts* :: $\langle ('a, 'b) \text{ form} \Rightarrow 'a \text{ list} \Rightarrow ('a, 'b) \text{ form} \rangle$ **where**
 $\langle \text{vars-for-consts } p [] = p \rangle$ |
 $\langle \text{vars-for-consts } p (c \# cs) = \text{subc } c (\text{Var } (\text{length } cs)) (\text{vars-for-consts } p cs) \rangle$

lemma *vars-for-consts*:
assumes $\langle \text{infinite } (- \text{ params } p) \rangle$
shows $\langle [] \vdash p \Longrightarrow [] \vdash \text{vars-for-consts } p xs \rangle$
 $\langle \text{proof} \rangle$

lemma *vars-for-consts-for-unis*:
 $\langle \text{closed } (\text{length } cs) p \Longrightarrow \text{list-all } (\lambda c. \text{new } c p) cs \Longrightarrow \text{distinct } cs \Longrightarrow$
 $\text{vars-for-consts } (\text{consts-for-unis } (\text{put-unis } (\text{length } cs) p) cs) cs = p \rangle$
 $\langle \text{proof} \rangle$

lemma *fresh-constant*:
fixes $p :: \langle ('a, 'b) \text{ form} \rangle$
assumes $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$
shows $\langle \exists c. c \notin \text{set } cs \wedge \text{new } c p \rangle$
 $\langle \text{proof} \rangle$

lemma *fresh-constants*:
fixes $p :: \langle ('a, 'b) \text{ form} \rangle$
assumes $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$
shows $\langle \exists cs. \text{length } cs = m \wedge \text{list-all } (\lambda c. \text{new } c p) cs \wedge \text{distinct } cs \rangle$
 $\langle \text{proof} \rangle$

lemma *closed-max*:
assumes $\langle \text{closed } m p \rangle \langle \text{closed } n q \rangle$
shows $\langle \text{closed } (\text{max } m n) p \wedge \text{closed } (\text{max } m n) q \rangle$
 $\langle \text{proof} \rangle$

lemma *ex-closed' [simp]*:
fixes $t :: \langle 'a \text{ term} \rangle$ **and** $l :: \langle 'a \text{ term list} \rangle$
shows $\langle \exists m. \text{closedt } m t \rangle \langle \exists n. \text{closedts } n l \rangle$
 $\langle \text{proof} \rangle$

lemma *ex-closed [simp]*: $\langle \exists m. \text{closed } m p \rangle$
 $\langle \text{proof} \rangle$

lemma *ex-closure*: $\langle \exists m. \text{closed } 0 (\text{put-unis } m p) \rangle$

$\langle proof \rangle$

lemma *remove-unis-sentence*:

assumes *inf-params*: $\langle infinite (- params p) \rangle$

and $\langle closed 0 (put-unis m p) \rangle \langle [] \vdash put-unis m p \rangle$

shows $\langle [] \vdash p \rangle$

$\langle proof \rangle$

9.6 Completeness

theorem *completeness*:

fixes $p :: \langle (nat, nat) form \rangle$

assumes $\langle \forall (e :: nat \Rightarrow nat hterm) f g. e, f, g, z \models p \rangle$

shows $\langle z \vdash p \rangle$

$\langle proof \rangle$

abbreviation $\langle valid p \equiv \forall (e :: nat \Rightarrow nat hterm) f g. eval e f g p \rangle$

proposition

fixes $p :: \langle (nat, nat) form \rangle$

shows $\langle valid p \Longrightarrow eval e f g p \rangle$

$\langle proof \rangle$

proposition

fixes $p :: \langle (nat, nat) form \rangle$

shows $\langle ([] \vdash p) = valid p \rangle$

$\langle proof \rangle$

corollary $\langle \forall e (f :: nat \Rightarrow nat hterm list \Rightarrow nat hterm) (g :: nat \Rightarrow nat hterm list \Rightarrow bool).$

$e, f, g, ps \models p \Longrightarrow ps \vdash p \rangle$

$\langle proof \rangle$

References

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