Meta-theory of first-order predicate logic

Stefan Berghofer

September 4, 2026

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*]: ‹$- A \cap B = B - A$› ⟨*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**

```
  ‹size-form FF = 0›
| ‹size-form TT = 0›
| ‹size-form (Pred - -) = 0›
| ‹size-form (And p q) = size-form p + size-form q + 1›
| ‹size-form (Or p q) = size-form p + size-form q + 1›
| ‹size-form (Impl p q) = size-form p + size-form q + 1›
| ‹size-form (Neg p) = size-form p + 1›
| ‹size-form (Forall p) = size-form p + 1›
| ‹size-form (Exists p) = size-form p + 1›
```

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
  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⟩

theorem closed-mono: assumes le: ‹i ≤ j›
  shows ‹closed i p ⟹ closed j p›
  ⟨proof⟩
```

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
  substt :: ‹*'a term* ⇒ *'a term* ⇒ *nat* ⇒ *'a term*› (‹-[-'/-]› [*300*, *0*, *0*] *300*) **and**
  substs :: ‹*'a term list* ⇒ *'a term* ⇒ *nat* ⇒ *'a term list*› (‹-[-'/-]› [*300*, *0*, *0*] *300*)
where
  ‹(*Var i*)[*s*/*k*] = (*if k* < *i then Var* (*i* − *1*) *else if i* = *k then s else Var i*)›
| ‹(*App a ts*)[*s*/*k*] = *App a* (*ts*[*s*/*k*])›
| ‹[][*s*/*k*] = []›
| ‹(*t* # *ts*)[*s*/*k*] = *t*[*s*/*k*] # *ts*[*s*/*k*]›

primrec
  liftt :: ‹*'a term* ⇒ *'a term*› **and**
  liftts :: ‹*'a term list* ⇒ *'a term list*› **where**
  ‹*liftt* (*Var i*) = *Var* (*Suc i*)›
| ‹*liftt* (*App a ts*) = *App a* (*liftts ts*)›
| ‹*liftts* [] = []›
| ‹*liftts* (*t* # *ts*) = *liftt t* # *liftts ts*›

primrec *subst* :: ‹(*'a*, *'b*) *form* ⇒ *'a term* ⇒ *nat* ⇒ (*'a*, *'b*) *form*›
  (‹-[-'/-]› [*300*, *0*, *0*] *300*) **where**
  ‹*FF*[*s*/*k*] = *FF*›
| ‹*TT*[*s*/*k*] = *TT*›
| ‹(*Pred b ts*)[*s*/*k*] = *Pred b* (*ts*[*s*/*k*])›
| ‹(*And p q*)[*s*/*k*] = *And* (*p*[*s*/*k*]) (*q*[*s*/*k*])›
| ‹(*Or p q*)[*s*/*k*] = *Or* (*p*[*s*/*k*]) (*q*[*s*/*k*])›
| ‹(*Impl p q*)[*s*/*k*] = *Impl* (*p*[*s*/*k*]) (*q*[*s*/*k*])›
| ‹(*Neg p*)[*s*/*k*] = *Neg* (*p*[*s*/*k*])›
| ‹(*Forall p*)[*s*/*k*] = *Forall* (*p*[*liftt s*/*Suc k*])›
| ‹(*Exists p*)[*s*/*k*] = *Exists* (*p*[*liftt s*/*Suc k*])›

theorem *lift-closed* [*simp*]:
  ‹*closedt 0* (*t*::*'a term*) ⟹ *closedt 0* (*liftt t*)›
  ‹*closedts 0* (*ts*::*'a term list*) ⟹ *closedts 0* (*liftts ts*)›
  ⟨*proof*⟩

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

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

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

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

primrec params :: ‹('a, 'b) form ⇒ 'a set› where
  ‹params FF = {}›
| ‹params TT = {}›
| ‹params (Pred b ts) = paramsts ts›
| ‹params (And p q) = params p ∪ params q›
| ‹params (Or p q) = params p ∪ params q›
| ‹params (Impl p q) = params p ∪ params q›
| ‹params (Neg p) = params p›
| ‹params (Forall p) = params p›
| ‹params (Exists p) = params p›
```

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

```
primrec
  psubstt :: ‹('a ⇒ 'c) ⇒ 'a term ⇒ 'c term› and
  psubstts :: ‹('a ⇒ 'c) ⇒ 'a term list ⇒ 'c term list› where
  ‹psubstt f (Var i) = Var i›
| ‹psubstt f (App x ts) = App (f x) (psubstts f ts)›
| ‹psubstts f [] = []›
| ‹psubstts f (t # ts) = psubstt f t # psubstts f ts›

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

theorem *psubstt-closed* [*simp*]:
‹*closedt i* (*psubstt f t*) = *closedt i t*›
‹*closedts i* (*psubstts f ts*) = *closedts i ts*›
⟨*proof*⟩

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

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

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

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

theorem *psubstt-upd* [*simp*]:
‹*x* ∉ *paramst* (*t*::'*a term*) ⟹ *psubstt* (*f*(*x* := *y*)) *t* = *psubstt f t*›
‹*x* ∉ *paramsts* (*ts*::'*a term list*) ⟹ *psubstts* (*f*(*x* := *y*)) *ts* = *psubstts f ts*›
⟨*proof*⟩

theorem *psubst-upd* [*simp*]: ‹*x* ∉ *params P* ⟹ *psubst* (*f*(*x* := *y*)) *P* = *psubst f P*›
⟨*proof*⟩

theorem *psubstt-id*:
fixes *t* :: ‹'*a term*› **and** *ts* :: ‹'*a term list*›
shows ‹*psubstt id t* = *t*› **and** ‹*psubstts* (λ*x*. *x*) *ts* = *ts*›
⟨*proof*⟩

theorem *psubst-id* [*simp*]: ‹*psubst id* = *id*›
⟨*proof*⟩

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

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

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* :: ‹(*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a*› (‹-⟨-:-⟩› [*90*, *0*, *0*] *91*)
where
  ‹*e*⟨*i*:*a*⟩ = (λ*j*. *if j* < *i then e j else if j* = *i then a else e* (*j* − *1*))›

lemma *shift-eq* [*simp*]: ‹*i* = *j* ⟹ (*e*⟨*i*:*T*⟩) *j* = *T*›
  ⟨*proof*⟩

lemma *shift-gt* [*simp*]: ‹*j* < *i* ⟹ (*e*⟨*i*:*T*⟩) *j* = *e j*›
  ⟨*proof*⟩

lemma *shift-lt* [*simp*]: ‹*i* < *j* ⟹ (*e*⟨*i*:*T*⟩) *j* = *e* (*j* − *1*)›
  ⟨*proof*⟩

lemma *shift-commute* [*simp*]: ‹*e*⟨*i*:*U*⟩⟨*0*:*T*⟩ = *e*⟨*0*:*T*⟩⟨*Suc i*:*U*⟩›
⟨*proof*⟩

primrec
  evalt :: ‹(*nat* ⇒ *'c*) ⇒ (*'a* ⇒ *'c list* ⇒ *'c*) ⇒ *'a term* ⇒ *'c*› **and**
  evalts :: ‹(*nat* ⇒ *'c*) ⇒ (*'a* ⇒ *'c list* ⇒ *'c*) ⇒ *'a term list* ⇒ *'c list*› **where**
  ‹*evalt e f* (*Var n*) = *e n*›
| ‹*evalt e f* (*App a ts*) = *f a* (*evalts e f ts*)›
| ‹*evalts e f* [] = []›
| ‹*evalts e f* (*t* # *ts*) = *evalt e f t* # *evalts e f ts*›

primrec *eval* :: ‹(*nat* ⇒ *'c*) ⇒ (*'a* ⇒ *'c list* ⇒ *'c*) ⇒
  (*'b* ⇒ *'c list* ⇒ *bool*) ⇒ (*'a*, *'b*) *form* ⇒ *bool*› **where**
  ‹*eval e f g FF* = *False*›
| ‹*eval e f g TT* = *True*›
| ‹*eval e f g* (*Pred a ts*) = *g a* (*evalts e f ts*)›
| ‹*eval e f g* (*And p q*) = ((*eval e f g p*) ∧ (*eval e f g q*))›
| ‹*eval e f g* (*Or p q*) = ((*eval e f g p*) ∨ (*eval e f g q*))›
| ‹*eval e f g* (*Impl p q*) = ((*eval e f g p*) ⟶ (*eval e f g q*))›
| ‹*eval e f g* (*Neg p*) = (¬ (*eval e f g p*))›
| ‹*eval e f g* (*Forall p*) = (∀ *z*. *eval* (*e*⟨*0*:*z*⟩) *f g p*)›
| ‹*eval e f g* (*Exists p*) = (∃ *z*. *eval* (*e*⟨*0*:*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 ps with respect to an environment e and interpretations f and g for function and predicate symbols, respectively.

definition *model* :: ‹(*nat* ⇒ *'c*) ⇒ (*'a* ⇒ *'c list* ⇒ *'c*) ⇒ (*'b* ⇒ *'c list* ⇒ *bool*) ⇒

('a, 'b) *form list* ⇒ ('a, 'b) *form* ⇒ *bool*› (‹-,-,-,- ⊨ -› [*50*,*50*] *50*) **where**
‹(*e*,*f*,*g*,*ps* ⊨ *p*) = (*list-all* (*eval e f g*) *ps* ⟶ *eval e f g p*)›

The following substitution lemmas relate substitution and evaluation functions:

theorem *subst-lemma'* [*simp*]:
‹*evalt e f* (*substt t u i*) = *evalt* (*e*⟨*i*:*evalt e f u*⟩) *f t*›
‹*evalts e f* (*substs ts u i*) = *evalts* (*e*⟨*i*:*evalt e f u*⟩) *f ts*›
⟨*proof*⟩

theorem *lift-lemma* [*simp*]:
‹*evalt* (*e*⟨*0*:*z*⟩) *f* (*liftt t*) = *evalt e f t*›
‹*evalts* (*e*⟨*0*:*z*⟩) *f* (*liftts ts*) = *evalts e f ts*›
⟨*proof*⟩

theorem *subst-lemma* [*simp*]:
‹*eval e f g* (*subst a t i*) = *eval* (*e*⟨*i*:*evalt e f t*⟩) *f g a*›
⟨*proof*⟩

theorem *upd-lemma'* [*simp*]:
‹*n* ∉ *paramst t* ⟹ *evalt e* (*f*(*n* := *x*)) *t* = *evalt e f t*›
‹*n* ∉ *paramsts ts* ⟹ *evalts e* (*f*(*n* := *x*)) *ts* = *evalts e f ts*›
⟨*proof*⟩

theorem *upd-lemma* [*simp*]:
‹*n* ∉ *params p* ⟹ *eval e* (*f*(*n* := *x*)) *g p* = *eval e f g p*›
⟨*proof*⟩

theorem *list-upd-lemma* [*simp*]: ‹*list-all* (λ*p*. *n* ∉ *params p*) *G* ⟹
list-all (*eval e* (*f*(*n* := *x*)) *g*) *G* = *list-all* (*eval e f g*) *G*›
⟨*proof*⟩

theorem *psubst-eval'* [*simp*]:
‹*evalt e f* (*psubstt h t*) = *evalt e* (λ*p*. *f* (*h p*)) *t*›
‹*evalts e f* (*psubstts h ts*) = *evalts e* (λ*p*. *f* (*h p*)) *ts*›
⟨*proof*⟩

theorem *psubst-eval*:
‹*eval e f g* (*psubst h p*) = *eval e* (λ*p*. *f* (*h p*)) *g p*›
⟨*proof*⟩

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

theorem *ex-all-commute-eval*:
‹*eval e f g* (*Impl* (*Exists* (*Forall* (*Pred p* [*Var 1*, *Var 0*])))
(*Forall* (*Exists* (*Pred p* [*Var 0*, *Var 1*])))))›
⟨*proof*⟩

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* :: ‹('a, 'b) *form list* ⇒ ('a, 'b) *form* ⇒ *bool*› (‹- ⊢ -› [*50*,*50*] *50*)
where
  Assum: ‹$a \in$ *set* $G \Longrightarrow G \vdash a$›
| *TTI*: ‹$G \vdash$ *TT*›
| *FFE*: ‹$G \vdash$ *FF* $\Longrightarrow G \vdash a$›
| *NegI*: ‹$a \# G \vdash$ *FF* $\Longrightarrow G \vdash$ *Neg* a›
| *NegE*: ‹$G \vdash$ *Neg* $a \Longrightarrow G \vdash a \Longrightarrow G \vdash$ *FF*›
| *Class*: ‹*Neg* $a \# G \vdash$ *FF* $\Longrightarrow G \vdash a$›
| *AndI*: ‹$G \vdash a \Longrightarrow G \vdash b \Longrightarrow G \vdash$ *And* $a\ b$›
| *AndE1*: ‹$G \vdash$ *And* $a\ b \Longrightarrow G \vdash a$›
| *AndE2*: ‹$G \vdash$ *And* $a\ b \Longrightarrow G \vdash b$›
| *OrI1*: ‹$G \vdash a \Longrightarrow G \vdash$ *Or* $a\ b$›
| *OrI2*: ‹$G \vdash b \Longrightarrow G \vdash$ *Or* $a\ b$›
| *OrE*: ‹$G \vdash$ *Or* $a\ b \Longrightarrow a \# G \vdash c \Longrightarrow b \# G \vdash c \Longrightarrow G \vdash c$›
| *ImplI*: ‹$a \# G \vdash b \Longrightarrow G \vdash$ *Impl* $a\ b$›
| *ImplE*: ‹$G \vdash$ *Impl* $a\ b \Longrightarrow G \vdash a \Longrightarrow G \vdash b$›
| *ForallI*: ‹$G \vdash a[$*App* $n\ [] / 0] \Longrightarrow$ *list-all* ($\lambda p.\ n \notin$ *params* p) $G \Longrightarrow$
    $n \notin$ *params* $a \Longrightarrow G \vdash$ *Forall* a›
| *ForallE*: ‹$G \vdash$ *Forall* $a \Longrightarrow G \vdash a[t/0]$›
| *ExistsI*: ‹$G \vdash a[t/0] \Longrightarrow G \vdash$ *Exists* a›
| *ExistsE*: ‹$G \vdash$ *Exists* $a \Longrightarrow a[$*App* $n\ [] / 0] \# G \vdash b \Longrightarrow$
    list-all ($\lambda p.\ n \notin$ *params* p) $G \Longrightarrow n \notin$ *params* $a \Longrightarrow n \notin$ *params* $b \Longrightarrow G \vdash b$›

The following derived inference rules are sometimes useful in applications.

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

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

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

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

theorem *tnd*: ‹$[] \vdash$ *Or* (*Pred* p $[]$) (*Neg* (*Pred* p $[]$))› (**is** ‹- ⊢ *?or*›)
⟨*proof*⟩

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

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

theorem *peirce*:
‹[] ⊢ *Impl* (*Impl* (*Impl* (*Pred P* []) (*Pred Q* [])) (*Pred P* [])) (*Pred P* [])›
(**is** ‹[] ⊢ *Impl ?PQP* (*Pred P* [])›)
⟨*proof*⟩

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

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

The latter property is called the *model existence theorem*. To see why 2 implies 1, observe that *Neg p*, $ps \nvdash FF$ 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. $S \nvdash FF$. 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 ϕ_0, ϕ_1, ϕ_2, ... containing all formulae. We can then

construct a sequence S_i of consistent sets as follows:

$$S_0 = S$$
$$S_{i+1} = \begin{cases} S_i \cup \{\phi_i\} & \text{if } S_i \cup \{\phi_i\} \text{ consistent} \\ S_i & \text{otherwise} \end{cases}$$

To obtain a maximal consistent set, we form the union $\bigcup_i S_i$ of these sets. To ensure that this union is still consistent, additional closure (see §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* :: ‹('*a*, '*b*) *form set set* ⇒ *bool*› **where**
‹*consistency C* = (∀ *S*. *S* ∈ *C* ⟶
(∀ *p ts*. ¬ (*Pred p ts* ∈ *S* ∧ *Neg* (*Pred p ts*) ∈ *S*)) ∧
FF ∉ *S* ∧ *Neg TT* ∉ *S* ∧
(∀ *Z*. *Neg* (*Neg Z*) ∈ *S* ⟶ *S* ∪ {*Z*} ∈ *C*) ∧
(∀ *A B*. *And A B* ∈ *S* ⟶ *S* ∪ {*A*, *B*} ∈ *C*) ∧
(∀ *A B*. *Neg* (*Or A B*) ∈ *S* ⟶ *S* ∪ {*Neg A*, *Neg B*} ∈ *C*) ∧
(∀ *A B*. *Or A B* ∈ *S* ⟶ *S* ∪ {*A*} ∈ *C* ∨ *S* ∪ {*B*} ∈ *C*) ∧
(∀ *A B*. *Neg* (*And A B*) ∈ *S* ⟶ *S* ∪ {*Neg A*} ∈ *C* ∨ *S* ∪ {*Neg B*} ∈ *C*) ∧
(∀ *A B*. *Impl A B* ∈ *S* ⟶ *S* ∪ {*Neg A*} ∈ *C* ∨ *S* ∪ {*B*} ∈ *C*) ∧
(∀ *A B*. *Neg* (*Impl A B*) ∈ *S* ⟶ *S* ∪ {*A*, *Neg B*} ∈ *C*) ∧
(∀ *P t*. *closedt 0 t* ⟶ *Forall P* ∈ *S* ⟶ *S* ∪ {*P*[*t*/*0*]} ∈ *C*) ∧
(∀ *P t*. *closedt 0 t* ⟶ *Neg* (*Exists P*) ∈ *S* ⟶ *S* ∪ {*Neg* (*P*[*t*/*0*])} ∈ *C*) ∧
(∀ *P*. *Exists P* ∈ *S* ⟶ (∃ *x*. *S* ∪ {*P*[*App x* []/*0*]} ∈ *C*)) ∧
(∀ *P*. *Neg* (*Forall P*) ∈ *S* ⟶ (∃ *x*. *S* ∪ {*Neg* (*P*[*App x* []/*0*])} ∈ *C*)))›

In §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* :: ‹('*a*, '*b*) *form set set* ⇒ *bool*› **where**
‹*alt-consistency C* = (∀ *S*. *S* ∈ *C* ⟶
(∀ *p ts*. ¬ (*Pred p ts* ∈ *S* ∧ *Neg* (*Pred p ts*) ∈ *S*)) ∧
FF ∉ *S* ∧ *Neg TT* ∉ *S* ∧
(∀ *Z*. *Neg* (*Neg Z*) ∈ *S* ⟶ *S* ∪ {*Z*} ∈ *C*) ∧
(∀ *A B*. *And A B* ∈ *S* ⟶ *S* ∪ {*A*, *B*} ∈ *C*) ∧
(∀ *A B*. *Neg* (*Or A B*) ∈ *S* ⟶ *S* ∪ {*Neg A*, *Neg B*} ∈ *C*) ∧
(∀ *A B*. *Or A B* ∈ *S* ⟶ *S* ∪ {*A*} ∈ *C* ∨ *S* ∪ {*B*} ∈ *C*) ∧
(∀ *A B*. *Neg* (*And A B*) ∈ *S* ⟶ *S* ∪ {*Neg A*} ∈ *C* ∨ *S* ∪ {*Neg B*} ∈ *C*) ∧
(∀ *A B*. *Impl A B* ∈ *S* ⟶ *S* ∪ {*Neg A*} ∈ *C* ∨ *S* ∪ {*B*} ∈ *C*) ∧

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

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

definition *mk-alt-consistency* :: ‹('a, 'b) *form set set* ⇒ ('a, 'b) *form set set*›
where
‹*mk-alt-consistency C* = {*S*. ∃*f*. *psubst f* ' *S* ∈ *C*}›

theorem *alt-consistency*:
assumes *conc*: ‹*consistency C*›
shows ‹*alt-consistency* (*mk-alt-consistency C*)› (**is** ‹*alt-consistency ?C'*›)
⟨*proof*⟩

theorem *mk-alt-consistency-subset*: ‹*C* ⊆ *mk-alt-consistency C*›
⟨*proof*⟩

7.2 Closure under subsets

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

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

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

lemma *subset-in-close*:
assumes ‹*S* ⊆ *S'*›
shows ‹*S'* ∪ *x* ∈ *C* ⟶ *S* ∪ *x* ∈ *close C*›
⟨*proof*⟩

theorem *close-consistency*:
assumes *conc*: ‹*consistency C*›
shows ‹*consistency* (*close C*)›
⟨*proof*⟩

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

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

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

theorem *mk-alt-consistency-closed*:
assumes ‹*subset-closed* C›
shows ‹*subset-closed* (*mk-alt-consistency* C)›
⟨*proof*⟩

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

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

theorem *finite-alt-consistency*:
assumes *altconc*: ‹*alt-consistency* C›
and ‹*subset-closed* C›
shows ‹*alt-consistency* (*mk-finite-char* C)›
⟨*proof*⟩

theorem *finite-char*: ‹*finite-char* (*mk-finite-char* C)›
⟨*proof*⟩

theorem *finite-char-closed*: ‹*finite-char* $C \Longrightarrow$ *subset-closed* C›
⟨*proof*⟩

theorem *finite-char-subset*: ‹*subset-closed* $C \Longrightarrow C \subseteq$ *mk-finite-char* C›
⟨*proof*⟩

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 ‹*term*› :: (*countable*) *countable*
⟨*proof*⟩

instance *form* :: (*countable*, *countable*) *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* = (∀ *n*. *f n* ⊆ *f* (*Suc n*))›

theorem *is-chainD*: ‹*is-chain f* ⟹ *x* ∈ *f m* ⟹ *x* ∈ *f* (*m* + *n*)›
⟨*proof*⟩

theorem *is-chainD'*:
assumes ‹*is-chain f*› **and** ‹*x* ∈ *f m*› **and** ‹*m* ≤ *k*›
shows ‹*x* ∈ *f k*›
⟨*proof*⟩

theorem *chain-index*:
assumes *ch*: ‹*is-chain f*› **and** *fin*: ‹*finite F*›
shows ‹*F* ⊆ (⋃ *n*. *f n*) ⟹ ∃ *n*. *F* ⊆ *f n*›
⟨*proof*⟩

lemma *chain-union-closed'*:
assumes ‹*is-chain f*› **and** ‹(∀ *n*. *f n* ∈ *C*)› **and** ‹∀ *S'* ∈ *C*. ∀ *S* ⊆ *S'*. *S* ∈ *C*›
and ‹*finite S'*› **and** ‹*S'* ⊆ (⋃ *n*. *f n*)›
shows ‹*S'* ∈ *C*›
⟨*proof*⟩

theorem *chain-union-closed*:
assumes ‹*finite-char C*› **and** ‹*is-chain f*› **and** ‹∀ *n*. *f n* ∈ *C*›
shows ‹(⋃ *n*. *f n*) ∈ *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*›

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

definition Extend :: ‹(nat, 'b) form set ⇒ (nat, 'b) form set set ⇒
    (nat ⇒ (nat, 'b) form) ⇒ (nat, 'b) form set› where
  ‹Extend S C f = (⋃ n. extend S C f n)›

theorem is-chain-extend: ‹is-chain (extend S C f)›
  ⟨proof⟩

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

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

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

lemma infinite-params-available:
  assumes ‹infinite (− (⋃ p ∈ S. params p))›
  shows ‹∃ x. x ∉ (⋃ p ∈ extend S C f n ∪ {f n}. params p)›
⟨proof⟩

lemma extend-in-C-Exists:
  assumes ‹alt-consistency C›
    and ‹infinite (− (⋃ p ∈ S. params p))›
    and ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
    and ‹∃ p. f n = Exists p›
  shows ‹extend S C f (Suc n) ∈ C›
⟨proof⟩

lemma extend-in-C-Neg-Forall:
  assumes ‹alt-consistency C›
    and ‹infinite (− (⋃ p ∈ S. params p))›
    and ‹extend S C f n ∪ {f n} ∈ C› (is ‹?S' ∈ C›)
    and ‹∀ p. f n ≠ Exists p›
    and ‹∃ p. f n = Neg (Forall p)›
  shows ‹extend S C f (Suc n) ∈ C›
```

⟨*proof*⟩

lemma *extend-in-C-no-delta*:
assumes ‹*extend S C f n* ∪ {*f n*} ∈ *C*›
and ‹∀ *p*. *f n* ≠ *Exists p*›
and ‹∀ *p*. *f n* ≠ *Neg* (*Forall p*)›
shows ‹*extend S C f* (*Suc n*) ∈ *C*›
⟨*proof*⟩

lemma *extend-in-C-stop*:
assumes ‹*extend S C f n* ∈ *C*›
and ‹*extend S C f n* ∪ {*f n*} ∉ *C*›
shows ‹*extend S C f* (*Suc n*) ∈ *C*›
⟨*proof*⟩

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

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

theorem *Extend-in-C*: ‹*alt-consistency C* ⟹ *finite-char C* ⟹
S ∈ *C* ⟹ *infinite* (− (⋃ *p* ∈ *S*. *params p*)) ⟹ *Extend S C f* ∈ *C*›
⟨*proof*⟩

theorem *Extend-subset*: ‹*S* ⊆ *Extend S C f*›
⟨*proof*⟩

The *Extend* function yields a maximal set:

definition *maximal* :: ‹'*a set* ⇒ '*a set set* ⇒ *bool*› **where**
‹*maximal S C* = (∀ *S'* ∈ *C*. *S* ⊆ *S'* ⟶ *S* = *S'*)›

theorem *extend-maximal*:
assumes ‹∀ *y*. ∃ *n*. *y* = *f n*›
and ‹*finite-char C*›
shows ‹*maximal* (*Extend S C f*) *C*›
⟨*proof*⟩

7.6 Hintikka sets and Herbrand models

A Hintikka set is defined as follows:

definition *hintikka* :: ‹('*a*, '*b*) *form set* ⇒ *bool*› **where**
‹*hintikka H* =
((∀ *p ts*. ¬ (*Pred p ts* ∈ *H* ∧ *Neg* (*Pred p ts*) ∈ *H*)) ∧
FF ∉ *H* ∧ *Neg TT* ∉ *H* ∧
(∀ *Z*. *Neg* (*Neg Z*) ∈ *H* ⟶ *Z* ∈ *H*) ∧
(∀ *A B*. *And A B* ∈ *H* ⟶ *A* ∈ *H* ∧ *B* ∈ *H*) ∧

$(\forall A\ B.\ Neg\ (Or\ A\ B) \in H \longrightarrow Neg\ A \in H \wedge Neg\ B \in H) \wedge$
$(\forall A\ B.\ Or\ A\ B \in H \longrightarrow A \in H \vee B \in H) \wedge$
$(\forall A\ B.\ Neg\ (And\ A\ B) \in H \longrightarrow Neg\ A \in H \vee Neg\ B \in H) \wedge$
$(\forall A\ B.\ Impl\ A\ B \in H \longrightarrow Neg\ A \in H \vee B \in H) \wedge$
$(\forall A\ B.\ Neg\ (Impl\ A\ B) \in H \longrightarrow A \in H \wedge Neg\ B \in H) \wedge$
$(\forall P\ t.\ closedt\ 0\ t \longrightarrow Forall\ P \in H \longrightarrow subst\ P\ t\ 0 \in H) \wedge$
$(\forall P\ t.\ closedt\ 0\ t \longrightarrow Neg\ (Exists\ P) \in H \longrightarrow Neg\ (subst\ P\ t\ 0) \in H) \wedge$
$(\forall P.\ Exists\ P \in H \longrightarrow (\exists t.\ closedt\ 0\ t \wedge subst\ P\ t\ 0 \in H)) \wedge$
$(\forall P.\ Neg\ (Forall\ P) \in H \longrightarrow (\exists t.\ closedt\ 0\ t \wedge Neg\ (subst\ P\ t\ 0) \in H)))$

In Herbrand models, each *closed* term is interpreted by itself. We introduce a new datatype *hterm* ("Herbrand terms"), which is similar to the datatype *term* introduced in §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* ∈ *H* ⟶ *closed 0 p* ⟶
eval e HApp (λ*a ts*. *Pred a* (*terms-of-hterms ts*) ∈ *H*) (*Neg p*))›
⟨*proof*⟩

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

lemma *Exists-in-extend*:
assumes ‹*extend S C f n* ∪ {*f n*} ∈ *C*› (**is** ‹*?S'* ∈ *C*›)
and ‹*Exists P* = *f n*›
shows ‹*P*[(*App* (*SOME k*. *k* ∉ (⋃*p* ∈ *extend S C f n* ∪ {*f n*}. *params p*)) [])/*0*]
∈
extend S C f (*Suc n*)›
(**is** ‹*subst P ?t 0* ∈ *extend S C f* (*Suc n*)›)
⟨*proof*⟩

lemma *Neg-Forall-in-extend*:
assumes ‹*extend S C f n* ∪ {*f n*} ∈ *C*› (**is** ‹*?S'* ∈ *C*›)
and ‹*Neg* (*Forall P*) = *f n*›
shows ‹*Neg* (*P*[(*App* (*SOME k*. *k* ∉ (⋃*p* ∈ *extend S C f n* ∪ {*f n*}. *params p*))
[])/*0*]) ∈
extend S C f (*Suc n*)›
(**is** ‹*Neg* (*subst P ?t 0*) ∈ *extend S C f* (*Suc n*)›)
⟨*proof*⟩

theorem *extend-hintikka*:
assumes *fin-ch*: ‹*finite-char C*›
and *infin-p*: ‹*infinite* (− (⋃*p* ∈ *S*. *params p*))›
and *surj*: ‹∀ *y*. ∃ *n*. *y* = *f n*›
and *altc*: ‹*alt-consistency C*›
and ‹*S* ∈ *C*›
shows ‹*hintikka* (*Extend S C f*)› (**is** ‹*hintikka ?H*›)
⟨*proof*⟩

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 ‹*consistency C*› **and** ‹*S* ∈ *C*›
and ‹*infinite* (− (⋃*p* ∈ *S*. *params p*))›
shows ‹*hintikka* (*Extend S* (*mk-finite-char* (*mk-alt-consistency* (*close C*))) *from-nat*)›
(**is** ‹*hintikka* (*Extend S ?C' from-nat*)›)
⟨*proof*⟩

theorem *model-existence*:
assumes ‹*consistency C*›
and ‹*S* ∈ *C*›
and ‹*infinite* (− (⋃*p* ∈ *S*. *params p*))›

and ‹$p \in S$›
and ‹*closed 0 p*›
shows ‹*eval e HApp* (λ*a ts. Pred a* (*terms-of-hterms ts*) $\in$ *Extend S*
(*mk-finite-char* (*mk-alt-consistency* (*close C*))) *from-nat*) *p*›
⟨*proof*⟩

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*: ‹*infinite* (*UNIV* :: *'a set*)›
shows ‹*consistency* {*S*::(*'a*, *'b*) *form set*. $\exists$ *G*. *S* = *set G* $\wedge$ $\neg$ *G* $\vdash$ *FF*}›
⟨*proof*⟩

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

theorem *natded-complete*:
assumes ‹*closed 0 p*›
and ‹*list-all* (*closed 0*) *ps*›
and *mod*: ‹$\forall$ *e f g*. *e*,(*f* :: *nat* $\Rightarrow$ *nat hterm list* $\Rightarrow$ *nat hterm*),
(*g* :: *nat* $\Rightarrow$ *nat hterm list* $\Rightarrow$ *bool*),*ps* $\models$ *p*›
shows ‹*ps* $\vdash$ *p*›
⟨*proof*⟩

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*:
‹*consistency* {*S*. *infinite* (− ($\bigcup$ *p* $\in$ *S*. *params p*)) $\wedge$ ($\exists$ *f*. $\forall$ (*p*::(*'a*, *'b*)*form*) $\in$ *S*.
eval e f g p)}›
⟨*proof*⟩

theorem *doublep-infinite-params*:
‹*infinite* (− ($\bigcup$ *p* $\in$ *psubst* (λ*n*::*nat*. *2* $*$ *n*) ‘ *S*. *params p*))›
⟨*proof*⟩

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

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

9 Completeness for open formulas

abbreviation ‹*new-term c t* ≡ *c* ∉ *paramst t*›
abbreviation ‹*new-list c ts* ≡ *c* ∉ *paramsts ts*›

abbreviation ‹*new c p* ≡ *c* ∉ *params p*›

abbreviation ‹*news c z* ≡ *list-all* (*new c*) *z*›

9.1 Renaming

lemma *new-psubst-image'*:
‹*new-term c t* ⟹ *d* ∉ *image f* (*paramst t*) ⟹ *new-term d* (*psubstt* (*f*(*c* := *d*)) *t*)›
‹*new-list c l* ⟹ *d* ∉ *image f* (*paramsts l*) ⟹ *new-list d* (*psubstts* (*f*(*c* := *d*)) *l*)›
⟨*proof*⟩

lemma *new-psubst-image*: ‹*new c p* ⟹ *d* ∉ *image f* (*params p*) ⟹ *new d* (*psubst* (*f*(*c* := *d*)) *p*)›
⟨*proof*⟩

lemma *news-psubst*: ‹*news c z* ⟹ *d* ∉ *image f* (⋃ *p* ∈ *set z*. *params p*) ⟹
news d (*map* (*psubst* (*f*(*c* := *d*))) *z*)›
⟨*proof*⟩

lemma *member-psubst*: ‹*p* ∈ *set z* ⟹ *psubst f p* ∈ *set* (*map* (*psubst f*) *z*)›
⟨*proof*⟩

lemma *deriv-psubst*:
fixes *f* :: ‹*'a* ⇒ *'a*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: *'a set*)›
shows ‹*z* ⊢ *p* ⟹ *map* (*psubst f*) *z* ⊢ *psubst f p*›
⟨*proof*⟩

9.2 Substitution for constants

primrec
subc-term :: ‹*'a* ⇒ *'a term* ⇒ *'a term* ⇒ *'a term*› **and**

subc-list :: ‹'*a* ⇒ '*a term* ⇒ '*a term list* ⇒ '*a term list*› **where**
‹*subc-term c s* (*Var n*) = *Var n*› |
‹*subc-term c s* (*App i l*) = (*if i* = *c then s else App i* (*subc-list c s l*))› |
‹*subc-list c s* [] = []› |
‹*subc-list c s* (*t* # *l*) = *subc-term c s t* # *subc-list c s l*›

primrec *subc* :: ‹'*a* ⇒ '*a term* ⇒ ('*a*, '*b*) *form* ⇒ ('*a*, '*b*) *form*› **where**
‹*subc c s FF* = *FF*› |
‹*subc c s TT* = *TT*› |
‹*subc c s* (*Pred i l*) = *Pred i* (*subc-list c s l*)› |
‹*subc c s* (*Neg p*) = *Neg* (*subc c s p*)› |
‹*subc c s* (*Impl p q*) = *Impl* (*subc c s p*) (*subc c s q*)› |
‹*subc c s* (*Or p q*) = *Or* (*subc c s p*) (*subc c s q*)› |
‹*subc c s* (*And p q*) = *And* (*subc c s p*) (*subc c s q*)› |
‹*subc c s* (*Exists p*) = *Exists* (*subc c* (*liftt s*) *p*)› |
‹*subc c s* (*Forall p*) = *Forall* (*subc c* (*liftt s*) *p*)›

primrec *subcs* :: ‹'*a* ⇒ '*a term* ⇒ ('*a*, '*b*) *form list* ⇒ ('*a*, '*b*) *form list*› **where**
‹*subcs c s* [] = []› |
‹*subcs c s* (*p* # *z*) = *subc c s p* # *subcs c s z*›

lemma *subst-0-lift*:
‹*substt* (*liftt t*) *s 0* = *t*›
‹*substs* (*liftts l*) *s 0* = *l*›
⟨*proof*⟩

lemma *params-lift* [*simp*]:
fixes *t* :: ‹'*a term*› **and** *ts* :: ‹'*a term list*›
shows
‹*paramst* (*liftt t*) = *paramst t*›
‹*paramsts* (*liftts ts*) = *paramsts ts*›
⟨*proof*⟩

lemma *subst-new'* [*simp*]:
‹*new-term c s* ⟹ *new-term c t* ⟹ *new-term c* (*substt t s m*)›
‹*new-term c s* ⟹ *new-list c l* ⟹ *new-list c* (*substs l s m*)›
⟨*proof*⟩

lemma *subst-new* [*simp*]: ‹*new-term c s* ⟹ *new c p* ⟹ *new c* (*subst p s m*)›
⟨*proof*⟩

lemma *subst-new-all*:
assumes ‹*a* ∉ *set cs*› ‹*list-all* (λ*c*. *new c p*) *cs*›
shows ‹*list-all* (λ*c*. *new c* (*subst p* (*App a* []) *m*)) *cs*›
⟨*proof*⟩

lemma *subc-new'* [*simp*]:
‹*new-term c t* ⟹ *subc-term c s t* = *t*›
‹*new-list c l* ⟹ *subc-list c s l* = *l*›

⟨*proof*⟩

lemma *subc-new* [*simp*]: ‹*new c p* ⟹ *subc c s p* = *p*›
⟨*proof*⟩

lemma *subcs-news*: ‹*news c z* ⟹ *subcs c s z* = *z*›
⟨*proof*⟩

lemma *subc-psubst′* [*simp*]:
‹(∀ *x* ∈ *paramst t*. *x* ≠ *c* ⟶ *f x* ≠ *f c*) ⟹
psubstt f (*subc-term c s t*) = *subc-term* (*f c*) (*psubstt f s*) (*psubstt f t*)›
‹(∀ *x* ∈ *paramsts l*. *x* ≠ *c* ⟶ *f x* ≠ *f c*) ⟹
psubstts f (*subc-list c s l*) = *subc-list* (*f c*) (*psubstt f s*) (*psubstts f l*)›
⟨*proof*⟩

lemma *subc-psubst*: ‹(∀ *x* ∈ *params p*. *x* ≠ *c* ⟶ *f x* ≠ *f c*) ⟹
psubst f (*subc c s p*) = *subc* (*f c*) (*psubstt f s*) (*psubst f p*)›
⟨*proof*⟩

lemma *subcs-psubst*: ‹(∀ *x* ∈ (⋃ *p* ∈ *set z*. *params p*). *x* ≠ *c* ⟶ *f x* ≠ *f c*) ⟹
map (*psubst f*) (*subcs c s z*) = *subcs* (*f c*) (*psubstt f s*) (*map* (*psubst f*) *z*)›
⟨*proof*⟩

lemma *new-lift*:
‹*new-term c t* ⟹ *new-term c* (*liftt t*)›
‹*new-list c l* ⟹ *new-list c* (*liftts l*)›
⟨*proof*⟩

lemma *new-subc′* [*simp*]:
‹*new-term d s* ⟹ *new-term d t* ⟹ *new-term d* (*subc-term c s t*)›
‹*new-term d s* ⟹ *new-list d l* ⟹ *new-list d* (*subc-list c s l*)›
⟨*proof*⟩

lemma *new-subc* [*simp*]: ‹*new-term d s* ⟹ *new d p* ⟹ *new d* (*subc c s p*)›
⟨*proof*⟩

lemma *news-subcs*: ‹*new-term d s* ⟹ *news d z* ⟹ *news d* (*subcs c s z*)›
⟨*proof*⟩

lemma *psubst-new-free′*:
‹*c* ≠ *n* ⟹ *new-term n* (*psubstt* (*id*(*n* := *c*)) *t*)›
‹*c* ≠ *n* ⟹ *new-list n* (*psubstts* (*id*(*n* := *c*)) *l*)›
⟨*proof*⟩

lemma *psubst-new-free*: ‹*c* ≠ *n* ⟹ *new n* (*psubst* (*id*(*n* := *c*)) *p*)›
⟨*proof*⟩

lemma *map-psubst-new-free*: ‹*c* ≠ *n* ⟹ *news n* (*map* (*psubst* (*id*(*n* := *c*))) *z*)›
⟨*proof*⟩

lemma *psubst-new-away'* [*simp*]:
‹*new-term fresh t* ⟹ *psubstt* (*id*(*fresh* := *c*)) (*psubstt* (*id*(*c* := *fresh*)) *t*) = *t*›
‹*new-list fresh l* ⟹ *psubstts* (*id*(*fresh* := *c*)) (*psubstts* (*id*(*c* := *fresh*)) *l*) = *l*›
⟨*proof*⟩

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

lemma *map-psubst-new-away*:
‹*news fresh z* ⟹ *map* (*psubst* (*id*(*fresh* := *c*))) (*map* (*psubst* (*id*(*c* := *fresh*))) *z*) = *z*›
⟨*proof*⟩

lemma *psubst-new'*:
‹*new-term c t* ⟹ *psubstt* (*id*(*c* := *x*)) *t* = *t*›
‹*new-list c l* ⟹ *psubstts* (*id*(*c* := *x*)) *l* = *l*›
⟨*proof*⟩

lemma *psubst-new*: ‹*new c p* ⟹ *psubst* (*id*(*c* := *x*)) *p* = *p*›
⟨*proof*⟩

lemma *map-psubst-new*: ‹*news c z* ⟹ *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* (*substs l u m*) = *substs* (*liftts l*) (*liftt u*) (*m* + *1*)›
⟨*proof*⟩

lemma *new-subc-same'* [*simp*]:
‹*new-term c s* ⟹ *new-term c* (*subc-term c s t*)›
‹*new-term c s* ⟹ *new-list c* (*subc-list c s l*)›
⟨*proof*⟩

lemma *new-subc-same*: ‹*new-term c s* ⟹ *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* ⟹ *subc-term c s* (*substt t u m*) = *subc-term c s* (*substt t* (*subc-term c s u*) *m*)›
‹*new-term c s* ⟹ *subc-list c s* (*substs l u m*) = *subc-list c s* (*substs l* (*subc-term c s u*) *m*)›

⟨*proof*⟩

lemma *new-subc-put*:
‹*new-term c s* ⟹ *subc c s* (*subst p t m*) = *subc c s* (*subst p* (*subc-term c s t*) *m*)›
⟨*proof*⟩

lemma *subc-subst-new'*:
‹*new-term c u* ⟹ *subc-term c* (*substt s u m*) (*substt t u m*) = *substt* (*subc-term c s t*) *u m*›
‹*new-term c u* ⟹ *subc-list c* (*substt s u m*) (*substs l u m*) = *substs* (*subc-list c s l*) *u m*›
⟨*proof*⟩

lemma *subc-subst-new*:
‹*new-term c t* ⟹ *subc c* (*substt s t m*) (*subst p t m*) = *subst* (*subc c s p*) *t m*›
⟨*proof*⟩

lemma *subc-sub-0-new* [*simp*]:
‹*new-term c t* ⟹ *subc c s* (*subst p t 0*) = *subst* (*subc c* (*liftt s*) *p*) *t 0*›
⟨*proof*⟩

lemma *member-subc*: ‹*p* ∈ *set z* ⟹ *subc c s p* ∈ *set* (*subcs c s z*)›
⟨*proof*⟩

lemma *deriv-subc*:
fixes *p* :: ‹('*a*, '*b*) *form*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
shows ‹*z* ⊢ *p* ⟹ *subcs c s z* ⊢ *subc c s p*›
⟨*proof*⟩

9.3 Weakening assumptions

lemma *psubst-new-subset*:
assumes ‹*set z* ⊆ *set z'*› ‹*c* ∉ (⋃ *p* ∈ *set z*. *params p*)›
shows ‹*set z* ⊆ *set* (*map* (*psubst* (*id*(*c* := *n*))) *z'*)›
⟨*proof*⟩

lemma *subset-cons*: ‹*set z* ⊆ *set z'* ⟹ *set* (*p* # *z*) ⊆ *set* (*p* # *z'*)›
⟨*proof*⟩

lemma *weaken-assumptions*:
fixes *p* :: ‹('*a*, '*b*) *form*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
shows ‹*z* ⊢ *p* ⟹ *set z* ⊆ *set z'* ⟹ *z'* ⊢ *p*›
⟨*proof*⟩

9.4 Implications and assumptions

primrec *put-imps* :: ‹('*a*, '*b*) *form* ⇒ ('*a*, '*b*) *form list* ⇒ ('*a*, '*b*) *form*› **where**

‹*put-imps p* [] = *p*› |
‹*put-imps p* (*q* # *z*) = *Impl q* (*put-imps p z*)›

lemma *semantics-put-imps*:
‹(*e*,*f*,*g*,*z* ⊨ *p*) = *eval e f g* (*put-imps p z*)›
⟨*proof*⟩

lemma *shift-imp-assum*:
fixes *p* :: ‹('*a*, '*b*) *form*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
and ‹*z* ⊢ *Impl p q*›
shows ‹*p* # *z* ⊢ *q*›
⟨*proof*⟩

lemma *remove-imps*:
assumes ‹*infinite* (− *params p*)›
shows ‹*z'* ⊢ *put-imps p z* ⟹ *rev z* @ *z'* ⊢ *p*›
⟨*proof*⟩

9.5 Closure elimination

lemma *subc-sub-closed-var'* [*simp*]:
‹*new-term c t* ⟹ *closedt* (*Suc m*) *t* ⟹ *subc-term c* (*Var m*) (*substt t* (*App c* []) *m*) = *t*›
‹*new-list c l* ⟹ *closedts* (*Suc m*) *l* ⟹ *subc-list c* (*Var m*) (*substs l* (*App c* []) *m*) = *l*›
⟨*proof*⟩

lemma *subc-sub-closed-var* [*simp*]: ‹*new c p* ⟹ *closed* (*Suc m*) *p* ⟹
subc c (*Var m*) (*subst p* (*App c* []) *m*) = *p*›
⟨*proof*⟩

primrec *put-unis* :: ‹*nat* ⇒ ('*a*, '*b*) *form* ⇒ ('*a*, '*b*) *form*› **where**
‹*put-unis* 0 *p* = *p*› |
‹*put-unis* (*Suc m*) *p* = *Forall* (*put-unis m p*)›

lemma *sub-put-unis* [*simp*]:
‹*subst* (*put-unis k p*) (*App c* []) *i* = *put-unis k* (*subst p* (*App c* []) (*i* + *k*))›
⟨*proof*⟩

lemma *closed-put-unis* [*simp*]: ‹*closed m* (*put-unis k p*) = *closed* (*m* + *k*) *p*›
⟨*proof*⟩

lemma *valid-put-unis*: ‹∀(*e* :: *nat* ⇒ '*a*) *f g*. *eval e f g p* ⟹
eval (*e* :: *nat* ⇒ '*a*) *f g* (*put-unis m p*)›
⟨*proof*⟩

lemma *put-unis-collapse*: ‹*put-unis m* (*put-unis n p*) = *put-unis* (*m* + *n*) *p*›
⟨*proof*⟩

fun *consts-for-unis* :: ‹('a, 'b) *form* ⇒ 'a *list* ⇒ ('a, 'b) *form*› **where**
‹*consts-for-unis* (*Forall p*) (*c*#*cs*) = *consts-for-unis* (*subst p* (*App c* []) *0*) *cs*› |
‹*consts-for-unis p* - = *p*›

lemma *consts-for-unis*: ‹[] ⊢ *put-unis* (*length cs*) *p* ⟹
[] ⊢ *consts-for-unis* (*put-unis* (*length cs*) *p*) *cs*›
⟨*proof*⟩

primrec *vars-for-consts* :: ‹('a, 'b) *form* ⇒ 'a *list* ⇒ ('a, 'b) *form*› **where**
‹*vars-for-consts p* [] = *p*› |
‹*vars-for-consts p* (*c* # *cs*) = *subc c* (*Var* (*length cs*)) (*vars-for-consts p cs*)›

lemma *vars-for-consts*:
assumes ‹*infinite* (− *params p*)›
shows ‹[] ⊢ *p* ⟹ [] ⊢ *vars-for-consts p xs*›
⟨*proof*⟩

lemma *vars-for-consts-for-unis*:
‹*closed* (*length cs*) *p* ⟹ *list-all* (λ*c*. *new c p*) *cs* ⟹ *distinct cs* ⟹
vars-for-consts (*consts-for-unis* (*put-unis* (*length cs*) *p*) *cs*) *cs* = *p*›
⟨*proof*⟩

lemma *fresh-constant*:
fixes *p* :: ‹('a, 'b) *form*›
assumes ‹*infinite* (*UNIV* :: 'a *set*)›
shows ‹∃ *c*. *c* ∉ *set cs* ∧ *new c p*›
⟨*proof*⟩

lemma *fresh-constants*:
fixes *p* :: ‹('a, 'b) *form*›
assumes ‹*infinite* (*UNIV* :: 'a *set*)›
shows ‹∃ *cs*. *length cs* = *m* ∧ *list-all* (λ*c*. *new c p*) *cs* ∧ *distinct cs*›
⟨*proof*⟩

lemma *closed-max*:
assumes ‹*closed m p*› ‹*closed n q*›
shows ‹*closed* (*max m n*) *p* ∧ *closed* (*max m n*) *q*›
⟨*proof*⟩

lemma *ex-closed'* [*simp*]:
fixes *t* :: ‹'a *term*› **and** *l* :: ‹'a *term list*›
shows ‹∃ *m*. *closedt m t*› ‹∃ *n*. *closedts n l*›
⟨*proof*⟩

lemma *ex-closed* [*simp*]: ‹∃ *m*. *closed m p*›
⟨*proof*⟩

lemma *ex-closure*: ‹∃ *m*. *closed 0* (*put-unis m p*)›

⟨*proof*⟩

lemma *remove-unis-sentence*:
assumes *inf-params*: ‹*infinite* (− *params p*)›
and ‹*closed 0* (*put-unis m p*)› ‹[] ⊢ *put-unis m p*›
shows ‹[] ⊢ *p*›
⟨*proof*⟩

9.6 Completeness

theorem *completeness*:
fixes *p* :: ‹(*nat*, *nat*) *form*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *e*, *f*, *g*, *z* ⊨ *p*›
shows ‹*z* ⊢ *p*›
⟨*proof*⟩

abbreviation ‹*valid p* ≡ ∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g p*›

proposition
fixes *p* :: ‹(*nat*, *nat*) *form*›
shows ‹*valid p* ⟹ *eval e f g p*›
⟨*proof*⟩

proposition
fixes *p* :: ‹(*nat*, *nat*) *form*›
shows ‹([] ⊢ *p*) = *valid p*›
⟨*proof*⟩

corollary ‹∀ *e* (*f*::*nat* ⇒ *nat hterm list* ⇒ *nat hterm*) (*g*::*nat* ⇒ *nat hterm list* ⇒ *bool*).
e,*f*,*g*,*ps* ⊨ *p* ⟹ *ps* ⊢ *p*›
⟨*proof*⟩

References

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