Meta-theory of first-order predicate logic

Stefan Berghofer

September 13, 2023

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 4
3.3 Parameters 5

4 Semantics **7**

5 Proof calculus **9**

6 Correctness **12**

7 Completeness **13**
7.1 Consistent sets 14
7.2 Closure under subsets 18
7.3 Finite character 22
7.4 Enumerating datatypes 29
7.5 Extension to maximal consistent sets 29
7.6 Hintikka sets and Herbrand models 34
7.7 Model existence theorem 44
7.8 Completeness for Natural Deduction 45

8 Löwenheim-Skolem theorem **54**

9 Completeness for open formulas **59**
9.1 Renaming 59
9.2 Substitution for constants 63
9.3 Weakening assumptions 74
9.4 Implications and assumptions 77
9.5 Closure elimination 78
9.6 Completeness 81

1 First-Order Logic According to Fitting

2 Miscellaneous Utilities

Some facts about (in)finite sets

theorem *set-inter-compl-diff* [*simp*]: ‹$- A \cap B = B - A$› **by** *blast*

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›
  using le by (induct t and ts rule: closedt.induct closedts.induct) simp-all

theorem closed-mono: assumes le: ‹i ≤ j›
  shows ‹closed i p ⟹ closed j p›
  using le
proof (induct p arbitrary: i j)
  case (Pred i l)
  then show ?case
    using closedt-mono by simp
qed auto
```

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

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

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

theorem *lift-closed* [*simp*]:
  ‹*closedt 0* (*t*::*'a term*) ⟹ *closedt 0* (*liftt t*)›
  ‹*closedts 0* (*ts*::*'a term list*) ⟹ *closedts 0* (*liftts ts*)›
  by (*induct t* **and** *ts rule*: *closedt.induct closedts.induct*) *simp-all*

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*])›
  using *u closedt-mono*(*1*)
  by (*induct t* **and** *ts rule*: *closedt.induct closedts.induct*) *auto*

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

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

theorem *subst-size-form* [*simp*]: ‹*size-form* (*subst p t i*) = *size-form p*›
by (*induct p arbitrary*: *i t*) *simp-all*

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*›
by (*induct t* **and** *ts rule*: *closedt.induct closedts.induct*) *simp-all*

theorem *psubst-closed* [*simp*]:
‹*closed i* (*psubst f p*) = *closed i p*›
by (*induct p arbitrary*: *i*) *simp-all*

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*›
by (*induct t* **and** *ts rule*: *psubstt.induct psubstts.induct*) *simp-all*

theorem *psubstt-lift* [*simp*]:
‹*psubstt f* (*liftt t*) = *liftt* (*psubstt f t*)›
‹*psubstts f* (*liftts ts*) = *liftts* (*psubstts f ts*)›
by (*induct t* **and** *ts rule*: *psubstt.induct psubstts.induct*) *simp-all*

theorem *psubst-subst* [*simp*]:
‹*psubst f* (*subst P t i*) = *subst* (*psubst f P*) (*psubstt f t*) *i*›
by (*induct P arbitrary*: *i t*) *simp-all*

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*›
by (*induct t* **and** *ts rule*: *psubstt.induct psubstts.induct*) (*auto split*: *sum.split*)

theorem *psubst-upd* [*simp*]: ‹*x* ∉ *params P* ⟹ *psubst* (*f*(*x* := *y*)) *P* = *psubst f P*›
by (*induct P*) (*simp-all del*: *fun-upd-apply*)

theorem *psubstt-id*:
fixes *t* :: ‹'*a term*› **and** *ts* :: ‹'*a term list*›
shows ‹*psubstt id t* = *t*› **and** ‹*psubstts* (λ*x*. *x*) *ts* = *ts*›
by (*induct t* **and** *ts rule*: *psubstt.induct psubstts.induct*) *simp-all*

theorem *psubst-id* [*simp*]: ‹*psubst id* = *id*›
proof
fix *p* :: ‹('*a*, '*b*) *form*›
show ‹*psubst id p* = *id p*›
by (*induct p*) (*simp-all add*: *psubstt-id*)
qed

theorem *psubstt-image* [*simp*]:
  ‹*paramst* (*psubstt f t*) = *f* ‘ *paramst t*›
  ‹*paramsts* (*psubstts f ts*) = *f* ‘ *paramsts ts*›
  by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) (*simp-all add*: *image-Un*)

theorem *psubst-image* [*simp*]: ‹*params* (*psubst f p*) = *f* ‘ *params p*›
  by (*induct p*) (*simp-all add*: *image-Un*)

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*›
  by (*simp add*: *shift-def*)

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

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

lemma *shift-commute* [*simp*]: ‹*e*⟨*i*:*U*⟩⟨*0*:*T*⟩ = *e*⟨*0*:*T*⟩⟨*Suc i*:*U*⟩›
proof
  fix *x*
  show ‹(*e*⟨*i*:*U*⟩⟨*0*:*T*⟩) *x* = (*e*⟨*0*:*T*⟩⟨*Suc i*:*U*⟩) *x*›
    by (*cases x*) (*simp-all add*: *shift-def*)
qed

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* ⊨ *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*›
by (*induct t* **and** *ts rule*: *substt.induct substs.induct*) *simp-all*

theorem *lift-lemma* [*simp*]:
‹*evalt* (*e*⟨*0*:*z*⟩) *f* (*liftt t*) = *evalt e f t*›
‹*evalts* (*e*⟨*0*:*z*⟩) *f* (*lifts ts*) = *evalts e f ts*›
by (*induct t* **and** *ts rule*: *liftt.induct lifts.induct*) *simp-all*

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

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*›
by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) *auto*

theorem *upd-lemma* [*simp*]:
‹*n* ∉ *params p* ⟹ *eval e* (*f*(*n* := *x*)) *g p* = *eval e f g p*›
by (*induct p arbitrary*: *e*) *simp-all*

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

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

by (*induct t* **and** *ts rule*: *psubstt.induct psubstts.induct*) *simp-all*

theorem *psubst-eval*:
‹*eval e f g* (*psubst h p*) = *eval e* (λ*p*. *f* (*h p*)) *g p*›
by (*induct p arbitrary*: *e*) *simp-all*

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

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

Simplification yields the following proof state:

1. (∃ *z*. ∀ *za*. *g p* [*z*, *za*]) ⟶ (∀ *z*. ∃ *za*. *g p* [*za*, *z*])

This is easily proved using intuitionistic logic:

by *iprover*

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* ∈ *set G* ⟹ *G* ⊢ *a*›
| *TTI*: ‹*G* ⊢ *TT*›
| *FFE*: ‹*G* ⊢ *FF* ⟹ *G* ⊢ *a*›
| *NegI*: ‹*a* # *G* ⊢ *FF* ⟹ *G* ⊢ *Neg a*›
| *NegE*: ‹*G* ⊢ *Neg a* ⟹ *G* ⊢ *a* ⟹ *G* ⊢ *FF*›
| *Class*: ‹*Neg a* # *G* ⊢ *FF* ⟹ *G* ⊢ *a*›
| *AndI*: ‹*G* ⊢ *a* ⟹ *G* ⊢ *b* ⟹ *G* ⊢ *And a b*›
| *AndE1*: ‹*G* ⊢ *And a b* ⟹ *G* ⊢ *a*›
| *AndE2*: ‹*G* ⊢ *And a b* ⟹ *G* ⊢ *b*›
| *OrI1*: ‹*G* ⊢ *a* ⟹ *G* ⊢ *Or a b*›
| *OrI2*: ‹*G* ⊢ *b* ⟹ *G* ⊢ *Or a b*›
| *OrE*: ‹*G* ⊢ *Or a b* ⟹ *a* # *G* ⊢ *c* ⟹ *b* # *G* ⊢ *c* ⟹ *G* ⊢ *c*›
| *ImplI*: ‹*a* # *G* ⊢ *b* ⟹ *G* ⊢ *Impl a b*›
| *ImplE*: ‹*G* ⊢ *Impl a b* ⟹ *G* ⊢ *a* ⟹ *G* ⊢ *b*›
| *ForallI*: ‹*G* ⊢ *a*[*App n* []/*0*] ⟹ *list-all* (λ*p*. *n* ∉ *params p*) *G* ⟹
n ∉ *params a* ⟹ *G* ⊢ *Forall a*›
| *ForallE*: ‹*G* ⊢ *Forall a* ⟹ *G* ⊢ *a*[*t*/*0*]›
| *ExistsI*: ‹*G* ⊢ *a*[*t*/*0*] ⟹ *G* ⊢ *Exists a*›
| *ExistsE*: ‹*G* ⊢ *Exists a* ⟹ *a*[*App n* []/*0*] # *G* ⊢ *b* ⟹
list-all (λ*p*. *n* ∉ *params p*) *G* ⟹ *n* ∉ *params a* ⟹ *n* ∉ *params b* ⟹ *G* ⊢ *b*›

The following derived inference rules are sometimes useful in applications.

theorem *Class'*: ‹*Neg A # G ⊢ A ⟹ G ⊢ A*›
by (*rule Class*, *rule NegE*, *rule Assum*) (*simp*, *iprover*)

theorem *cut*: ‹*G ⊢ A ⟹ A # G ⊢ B ⟹ G ⊢ B*›
by (*rule ImplE*, *rule ImplI*)

theorem *ForallE'*: ‹*G ⊢ Forall a ⟹ subst a t 0 # G ⊢ B ⟹ G ⊢ B*›
by (*rule cut*, *rule ForallE*)

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

theorem *tnd*: ‹[] ⊢ *Or* (*Pred p* []) (*Neg* (*Pred p* []))› (**is** ‹- ⊢ *?or*›)
proof −
have ‹[*Neg ?or*] ⊢ *Neg ?or*›
by (*simp add*: *Assum*)
moreover { **have** ‹[*Pred p* [], *Neg ?or*] ⊢ *Neg ?or*›
by (*simp add*: *Assum*)
moreover have ‹[*Pred p* [], *Neg ?or*] ⊢ *Pred p* []›
by (*simp add*: *Assum*)
then have ‹[*Pred p* [], *Neg ?or*] ⊢ *?or*›
by (*rule OrI1*)
ultimately have ‹[*Pred p* [], *Neg ?or*] ⊢ *FF*›
by (*rule NegE*)
then have ‹[*Neg ?or*] ⊢ *Neg* (*Pred p* [])›
by (*rule NegI*)
then have ‹[*Neg ?or*] ⊢ *?or*›
by (*rule OrI2*) }
ultimately have ‹[*Neg ?or*] ⊢ *FF*›
by (*rule NegE*)
then show *?thesis*
by (*rule Class*)
qed

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 −
let *?forall* = ‹*Forall* (*Pred p* [*Var 1*, *Var 0*]) :: (*nat*, *'b*) *form*›

have ‹[*Exists ?forall*] ⊢ *Exists ?forall*›
by (*simp add*: *Assum*)
moreover { **have** ‹[*?forall*[*App 1* []/*0*], *Exists ?forall*] ⊢ *Forall* (*Pred p* [*App 1* [], *Var 0*])›
by (*simp add*: *Assum*)
moreover have ‹[*Pred p* [*App 1* [], *Var 0*][*App 0* []/*0*], *?forall*[*App 1* []/*0*], *Exists ?forall*] ⊢ *Pred p* [*Var 0*, *App 0* []][*App 1* []/*0*]›
by (*simp add*: *Assum*)
ultimately have ‹[*?forall*[*App 1* []/*0*], *Exists ?forall*] ⊢ (*Pred p* [*Var 0*, *App 0*

```
[])[App 1 []/0]›
      by (rule ForallE') }
  then have ‹[?forall[App 1 []/0], Exists ?forall] ⊢ Exists (Pred p [Var 0, App 0
[])›
    by (rule ExistsI)
  moreover have ‹list-all (λp. 1 ∉ params p) [Exists ?forall]›
    by simp
  moreover have ‹1 ∉ params ?forall›
    by simp
  moreover have ‹1 ∉ params (Exists (Pred p [Var 0, App (0 :: nat) []]))›
    by simp
  ultimately have ‹[Exists ?forall] ⊢ Exists (Pred p [Var 0, App 0 []])›
    by (rule ExistsE)
  then have ‹[Exists ?forall] ⊢ (Exists (Pred p [Var 0, Var 1]))[App 0 []/0]›
    by simp
  moreover have ‹list-all (λp. 0 ∉ params p) [Exists ?forall]›
    by simp
  moreover have ‹0 ∉ params (Exists (Pred p [Var 0, Var 1]))›
    by simp
  ultimately have ‹[Exists ?forall] ⊢ Forall (Exists (Pred p [Var 0, Var 1]))›
    by (rule ForallI)
  then show ?thesis
    by (rule ImplI)
qed

theorem drinker: ‹([]::(nat, 'b) form list) ⊢
  Exists (Impl (Pred P [Var 0]) (Forall (Pred P [Var 0])))›
proof −
  let ?impl = ‹(Impl (Pred P [Var 0]) (Forall (Pred P [Var 0]))) :: (nat, 'b) form›
  let ?G' = ‹[Pred P [Var 0], Neg (Exists ?impl)]›
  let ?G = ‹Neg (Pred P [App 0 []]) # ?G'›

  have ‹?G ⊢ Neg (Exists ?impl)›
    by (simp add: Assum)
  moreover have ‹Pred P [App 0 []] # ?G ⊢ Neg (Pred P [App 0 []])›
    and ‹Pred P [App 0 []] # ?G ⊢ Pred P [App 0 []]›
    by (simp-all add: Assum)
  then have ‹Pred P [App 0 []] # ?G ⊢ FF›
    by (rule NegE)
  then have ‹Pred P [App 0 []] # ?G ⊢ Forall (Pred P [Var 0])›
    by (rule FFE)
  then have ‹?G ⊢ ?impl[App 0 []/0]›
    using ImplI by simp
  then have ‹?G ⊢ Exists ?impl›
    by (rule ExistsI)
  ultimately have ‹?G ⊢ FF›
    by (rule NegE)
  then have ‹?G' ⊢ Pred P [Var 0][App 0 []/0]›
    using Class by simp
```

```
  moreover have ‹list-all (λp. (0 :: nat) ∉ params p) ?G'›
    by simp
  moreover have ‹(0 :: nat) ∉ params (Pred P [Var 0])›
    by simp
  ultimately have ‹?G' ⊢ Forall (Pred P [Var 0])›
    by (rule ForallI)
  then have ‹[Neg (Exists ?impl)] ⊢ ?impl[Var 0/0]›
    using ImplI by simp
  then have ‹[Neg (Exists ?impl)] ⊢ Exists ?impl›
    by (rule ExistsI)
  then show ?thesis
    by (rule Class')
qed

theorem peirce:
  ‹[] ⊢ Impl (Impl (Impl (Pred P []) (Pred Q [])) (Pred P [])) (Pred P [])›
  (is ‹[] ⊢ Impl ?PQP (Pred P [])›)
proof −
  let ?PQPP = ‹Impl ?PQP (Pred P [])›

  have ‹[?PQP, Neg ?PQPP] ⊢ ?PQP›
    by (simp add: Assum)
  moreover { have ‹[Pred P [], ?PQP, Neg ?PQPP] ⊢ Neg ?PQPP›
      by (simp add: Assum)
    moreover have ‹[?PQP, Pred P [], ?PQP, Neg ?PQPP] ⊢ Pred P []›
      by (simp add: Assum)
    then have ‹[Pred P [], ?PQP, Neg ?PQPP] ⊢ ?PQPP›
      by (rule ImplI)
    ultimately have ‹[Pred P [], ?PQP, Neg ?PQPP] ⊢ FF›
      by (rule NegE) }
  then have ‹[Pred P [], ?PQP, Neg ?PQPP] ⊢ Pred Q []›
    by (rule FFE)
  then have ‹[?PQP, Neg ?PQPP] ⊢ Impl (Pred P []) (Pred Q [])›
    by (rule ImplI)
  ultimately have ‹[?PQP, Neg ?PQPP] ⊢ Pred P []›
    by (rule ImplE)
  then have ‹[Neg ?PQPP] ⊢ ?PQPP›
    by (rule ImplI)
  then show ‹[] ⊢ ?PQPP›
    by (rule Class')
qed
```

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 ⊢ p ⟹ ∀ e f g. e,f,g,G ⊨ p›
```

```
proof (induct p rule: deriv.induct)
  case (Assum a G)
  then show ?case by (simp add: model-def list-all-iff)
next
  case (ForallI G a n)
  show ?case
  proof (intro allI)
    fix f g and e :: ‹nat ⇒ 'c›
    have ‹∀ z. e, (f(n := λx. z)), g, G ⊨ (a[App n []/0])›
      using ForallI by blast
    then have ‹∀ z. list-all (eval e f g) G ⟶ eval (e⟨0:z⟩) f g a›
      using ForallI unfolding model-def by simp
    then show ‹e,f,g,G ⊨ Forall a› unfolding model-def by simp
  qed
next
  case (ExistsE G a n b)
  show ?case
  proof (intro allI)
    fix f g and e :: ‹nat ⇒ 'c›
    obtain z where ‹list-all (eval e f g) G ⟶ eval (e⟨0:z⟩) f g a›
      using ExistsE unfolding model-def by simp blast
    then have ‹e, (f(n := λx. z)), g, G ⊨ b›
      using ExistsE unfolding model-def by simp
    then show ‹e,f,g,G ⊨ b›
      using ExistsE unfolding model-def by simp
  qed
qed (simp-all add: model-def, blast+)
```

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*)) ∧

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

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'*›)
unfolding *alt-consistency-def*
proof (*intro allI impI conjI*)
fix *f* :: ‹'a ⇒ 'a› **and** *S* :: ‹('a, 'b) *form set*›

assume ‹*S* ∈ *mk-alt-consistency C*›
then obtain *f* **where** *sc*: ‹*psubst f* ‘ *S* ∈ *C*› (**is** ‹*?S'* ∈ *C*›)
unfolding *mk-alt-consistency-def* **by** *blast*

fix *p ts*
show ‹¬ (*Pred p ts* ∈ *S* ∧ *Neg* (*Pred p ts*) ∈ *S*)›
proof
assume ∗: ‹*Pred p ts* ∈ *S* ∧ *Neg* (*Pred p ts*) ∈ *S*›
then have ‹*psubst f* (*Pred p ts*) ∈ *?S'*›
by *blast*
then have ‹*Pred p* (*psubstts f ts*) ∈ *?S'*›
by *simp*
then have ‹*Neg* (*Pred p* (*psubstts f ts*)) ∉ *?S'*›
using *conc sc* **by** (*simp add*: *consistency-def*)
then have ‹*Neg* (*Pred p ts*) ∉ *S*›
by *force*
then show *False*
using ∗ **by** *blast*

qed

have ‹*FF* ∉ *?S′*› **and** ‹*Neg TT* ∉ *?S′*›
  using *conc sc* **unfolding** *consistency-def* **by** *simp-all*
then show ‹*FF* ∉ *S*› **and** ‹*Neg TT* ∉ *S*›
  by (*force*, *force*)

{ fix *Z*
  assume ‹*Neg* (*Neg Z*) ∈ *S*›
  then have ‹*psubst f* (*Neg* (*Neg Z*)) ∈ *?S′*›
    by *blast*
  then have ‹*?S′* ∪ {*psubst f Z*} ∈ *C*›
    using *conc sc* **by** (*simp add*: *consistency-def*)
  then show ‹*S* ∪ {*Z*} ∈ *?C′*›
    unfolding *mk-alt-consistency-def* **by** *auto* **}**

{ fix *A B*
  assume ‹*And A B* ∈ *S*›
  then have ‹*psubst f* (*And A B*) ∈ *?S′*›
    by *blast*
  then have ‹*?S′* ∪ {*psubst f A*, *psubst f B*} ∈ *C*›
    using *conc sc* **by** (*simp add*: *consistency-def*)
  then show ‹*S* ∪ {*A*, *B*} ∈ *?C′*›
    unfolding *mk-alt-consistency-def* **by** *auto* **}**

{ fix *A B*
  assume ‹*Neg* (*Or A B*) ∈ *S*›
  then have ‹*psubst f* (*Neg* (*Or A B*)) ∈ *?S′*›
    by *blast*
  then have ‹*?S′* ∪ {*Neg* (*psubst f A*), *Neg* (*psubst f B*)} ∈ *C*›
    using *conc sc* **by** (*simp add*: *consistency-def*)
  then show ‹*S* ∪ {*Neg A*, *Neg B*} ∈ *?C′*›
    unfolding *mk-alt-consistency-def* **by** *auto* **}**

{ fix *A B*
  assume ‹*Neg* (*Impl A B*) ∈ *S*›
  then have ‹*psubst f* (*Neg* (*Impl A B*)) ∈ *?S′*›
    by *blast*
  then have ‹*?S′* ∪ {*psubst f A*, *Neg* (*psubst f B*)} ∈ *C*›
    using *conc sc* **by** (*simp add*: *consistency-def*)
  then show ‹*S* ∪ {*A*, *Neg B*} ∈ *?C′*›
    unfolding *mk-alt-consistency-def* **by** *auto* **}**

{ fix *A B*
  assume ‹*Or A B* ∈ *S*›
  then have ‹*psubst f* (*Or A B*) ∈ *?S′*›
    by *blast*
  then have ‹*?S′* ∪ {*psubst f A*} ∈ *C* ∨ *?S′* ∪ {*psubst f B*} ∈ *C*›
    using *conc sc* **by** (*simp add*: *consistency-def*)

```
  then show ‹S ∪ {A} ∈ ?C' ∨ S ∪ {B} ∈ ?C'›
    unfolding mk-alt-consistency-def by auto }

{ fix A B
  assume ‹Neg (And A B) ∈ S›
  then have ‹psubst f (Neg (And A B)) ∈ ?S'›
    by blast
  then have ‹?S' ∪ {Neg (psubst f A)} ∈ C ∨ ?S' ∪ {Neg (psubst f B)} ∈ C›
    using conc sc by (simp add: consistency-def)
  then show ‹S ∪ {Neg A} ∈ ?C' ∨ S ∪ {Neg B} ∈ ?C'›
    unfolding mk-alt-consistency-def by auto }

{ fix A B
  assume ‹Impl A B ∈ S›
  then have ‹psubst f (Impl A B) ∈ ?S'›
    by blast
  then have ‹?S' ∪ {Neg (psubst f A)} ∈ C ∨ ?S' ∪ {psubst f B} ∈ C›
    using conc sc by (simp add: consistency-def)
  then show ‹S ∪ {Neg A} ∈ ?C' ∨ S ∪ {B} ∈ ?C'›
    unfolding mk-alt-consistency-def by auto }

{ fix P and t :: ‹'a term›
  assume ‹closedt 0 t› and ‹Forall P ∈ S›
  then have ‹psubst f (Forall P) ∈ ?S'›
    by blast
  then have ‹?S' ∪ {psubst f P[psubstt f t/0]} ∈ C›
    using ‹closedt 0 t› conc sc by (simp add: consistency-def)
  then show ‹S ∪ {P[t/0]} ∈ ?C'›
    unfolding mk-alt-consistency-def by auto }

{ fix P and t :: ‹'a term›
  assume ‹closedt 0 t› and ‹Neg (Exists P) ∈ S›
  then have ‹psubst f (Neg (Exists P)) ∈ ?S'›
    by blast
  then have ‹?S' ∪ {Neg (psubst f P[psubstt f t/0])} ∈ C›
    using ‹closedt 0 t› conc sc by (simp add: consistency-def)
  then show ‹S ∪ {Neg (P[t/0])} ∈ ?C'›
    unfolding mk-alt-consistency-def by auto }

{ fix P :: ‹('a, 'b) form› and x f'
  assume ‹∀ a ∈ S. x ∉ params a› and ‹Exists P ∈ S›
  moreover have ‹psubst f (Exists P) ∈ ?S'›
    using calculation by blast
  then have ‹∃ y. ?S' ∪ {psubst f P[App y []/0]} ∈ C›
    using conc sc by (simp add: consistency-def)
  then obtain y where ‹?S' ∪ {psubst f P[App y []/0]} ∈ C›
    by blast

  moreover have ‹psubst (f(x := y)) ‘ S = ?S'›
```

```
    using calculation by (simp cong add: image-cong)
   moreover have ‹psubst (f(x := y)) ‘
      S ∪ {psubst (f(x := y)) P[App ((f(x := y)) x) []/0]} ∈ C›
    using calculation by auto
   ultimately have ‹∃f. psubst f ‘ S ∪ {psubst f P[App (f x) []/0]} ∈ C›
    by blast
   then show ‹S ∪ {P[App x []/0]} ∈ ?C'›
    unfolding mk-alt-consistency-def by simp }

 { fix P :: ‹('a, 'b) form› and x
   assume ‹∀ a ∈ S. x ∉ params a› and ‹Neg (Forall P) ∈ S›
   moreover have ‹psubst f (Neg (Forall P)) ∈ ?S'›
    using calculation by blast
   then have ‹∃ y. ?S' ∪ {Neg (psubst f P[App y []/0])} ∈ C›
    using conc sc by (simp add: consistency-def)
   then obtain y where ‹?S' ∪ {Neg (psubst f P[App y []/0])} ∈ C›
    by blast

   moreover have ‹psubst (f(x := y)) ‘ S = ?S'›
    using calculation by (simp cong add: image-cong)
   moreover have ‹psubst (f(x := y)) ‘
   S ∪ {Neg (psubst (f(x := y)) P[App ((f(x := y)) x) []/0])} ∈ C›
    using calculation by auto
   ultimately have ‹∃f. psubst f ‘ S ∪ {Neg (psubst f P[App (f x) []/0])} ∈ C›
    by blast
   then show ‹S ∪ {Neg (P[App x []/0])} ∈ ?C'›
    unfolding mk-alt-consistency-def by simp }
qed

theorem mk-alt-consistency-subset: ‹C ⊆ mk-alt-consistency C›
 unfolding mk-alt-consistency-def
proof
 fix x assume ‹x ∈ C›
 then have ‹psubst id ‘ x ∈ C›
  by simp
 then have ‹(∃f. psubst f ‘ x ∈ C)›
  by blast
 then show ‹x ∈ {S. ∃f. psubst f ‘ S ∈ C}›
  by simp
qed
```

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 −
  have ‹S' ∪ x ∈ close C ⟶ S ∪ x ∈ close C›
    unfolding close-def using ‹S ⊆ S'› by blast
  then show ?thesis unfolding close-def by blast
qed

theorem close-consistency:
  assumes conc: ‹consistency C›
  shows ‹consistency (close C)›
  unfolding consistency-def
proof (intro allI impI conjI)
  fix S
  assume ‹S ∈ close C›
  then obtain x where ‹x ∈ C› and ‹S ⊆ x›
    unfolding close-def by blast

  { fix p ts
    have ‹¬ (Pred p ts ∈ x ∧ Neg (Pred p ts) ∈ x)›
      using ‹x ∈ C› conc unfolding consistency-def by simp
    then show ‹¬ (Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S)›
      using ‹S ⊆ x› by blast }

  { have ‹FF ∉ x›
      using ‹x ∈ C› conc unfolding consistency-def by blast
    then show ‹FF ∉ S›
      using ‹S ⊆ x› by blast }

  { have ‹Neg TT ∉ x›
      using ‹x ∈ C› conc unfolding consistency-def by blast
    then show ‹Neg TT ∉ S›
      using ‹S ⊆ x› by blast }

  { fix Z
    assume ‹Neg (Neg Z) ∈ S›
    then have ‹Neg (Neg Z) ∈ x›
      using ‹S ⊆ x› by blast
    then have ‹x ∪ {Z} ∈ C›
      using ‹x ∈ C› conc unfolding consistency-def by simp
    then show ‹S ∪ {Z} ∈ close C›
      using ‹S ⊆ x› subset-in-close by blast }

  { fix A B
    assume ‹And A B ∈ S›
    then have ‹And A B ∈ x›
```

```
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {A, B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by simp
  then show ‹S ∪ {A, B} ∈ close C›
    using ‹S ⊆ x› subset-in-close by blast }

{ fix A B
  assume ‹Neg (Or A B) ∈ S›
  then have ‹Neg (Or A B) ∈ x›
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {Neg A, Neg B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by simp
  then show ‹S ∪ {Neg A, Neg B} ∈ close C›
    using ‹S ⊆ x› subset-in-close by blast }

{ fix A B
  assume ‹Or A B ∈ S›
  then have ‹Or A B ∈ x›
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {A} ∈ C ∨ x ∪ {B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by simp
  then show ‹S ∪ {A} ∈ close C ∨ S ∪ {B} ∈ close C›
    using ‹S ⊆ x› subset-in-close by blast }

{ fix A B
  assume ‹Neg (And A B) ∈ S›
  then have ‹Neg (And A B) ∈ x›
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {Neg A} ∈ C ∨ x ∪ {Neg B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by simp
  then show ‹S ∪ {Neg A} ∈ close C ∨ S ∪ {Neg B} ∈ close C›
    using ‹S ⊆ x› subset-in-close by blast }

{ fix A B
  assume ‹Impl A B ∈ S›
  then have ‹Impl A B ∈ x›
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {Neg A} ∈ C ∨ x ∪ {B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by simp
  then show ‹S ∪ {Neg A} ∈ close C ∨ S ∪ {B} ∈ close C›
    using ‹S ⊆ x› subset-in-close by blast }

{ fix A B
  assume ‹Neg (Impl A B) ∈ S›
  then have ‹Neg (Impl A B) ∈ x›
    using ‹S ⊆ x› by blast
  then have ‹x ∪ {A, Neg B} ∈ C›
    using ‹x ∈ C› conc unfolding consistency-def by blast
  then show ‹S ∪ {A, Neg B} ∈ close C›
```

```
    using ‹S ⊆ x› subset-in-close by blast }

  { fix P and t :: ‹'a term›
    assume ‹closedt 0 t› and ‹Forall P ∈ S›
    then have ‹Forall P ∈ x›
      using ‹S ⊆ x› by blast
    then have ‹x ∪ {P[t/0]} ∈ C›
      using ‹closedt 0 t› ‹x ∈ C› conc unfolding consistency-def by blast
    then show ‹S ∪ {P[t/0]} ∈ close C›
      using ‹S ⊆ x› subset-in-close by blast }

  { fix P and t :: ‹'a term›
    assume ‹closedt 0 t› and ‹Neg (Exists P) ∈ S›
    then have ‹Neg (Exists P) ∈ x›
      using ‹S ⊆ x› by blast
    then have ‹x ∪ {Neg (P[t/0])} ∈ C›
      using ‹closedt 0 t› ‹x ∈ C› conc unfolding consistency-def by blast
    then show ‹S ∪ {Neg (P[t/0])} ∈ close C›
      using ‹S ⊆ x› subset-in-close by blast }

  { fix P
    assume ‹Exists P ∈ S›
    then have ‹Exists P ∈ x›
      using ‹S ⊆ x› by blast
    then have ‹∃ c. x ∪ {P[App c []/0]} ∈ C›
      using ‹x ∈ C› conc unfolding consistency-def by blast
    then show ‹∃ c. S ∪ {P[App c []/0]} ∈ close C›
      using ‹S ⊆ x› subset-in-close by blast }

  { fix P
    assume ‹Neg (Forall P) ∈ S›
    then have ‹Neg (Forall P) ∈ x›
      using ‹S ⊆ x› by blast
    then have ‹∃ c. x ∪ {Neg (P[App c []/0])} ∈ C›
      using ‹x ∈ C› conc unfolding consistency-def by simp
    then show ‹∃ c. S ∪ {Neg (P[App c []/0])} ∈ close C›
      using ‹S ⊆ x› subset-in-close by blast }
qed

theorem close-closed: ‹subset-closed (close C)›
  unfolding close-def subset-closed-def by blast

theorem close-subset: ‹C ⊆ close C›
  unfolding close-def by blast
```

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*)›
unfolding *subset-closed-def mk-alt-consistency-def*
proof (*intro ballI allI impI*)
fix *S S'* :: ‹('*a*, '*b*) *form set*›
assume ‹*S* ⊆ *S'*› **and** ‹*S'* ∈ {*S*. ∃ *f*. *psubst f* ' *S* ∈ *C*}›
then obtain *f* **where** ∗: ‹*psubst f* ' *S'* ∈ *C*›
by *blast*
moreover have ‹*psubst f* ' *S* ⊆ *psubst f* ' *S'*›
using ‹*S* ⊆ *S'*› **by** *blast*
moreover have ‹∀ *S'* ∈ *C*. ∀ *S* ⊆ *S'*. *S* ∈ *C*›
using ‹*subset-closed C*› **unfolding** *subset-closed-def* **by** *blast*
ultimately have ‹*psubst f* ' *S* ∈ *C*›
by *blast*
then show ‹*S* ∈ {*S*. ∃ *f*. *psubst f* ' *S* ∈ *C*}›
by *blast*
qed

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* = (∀ *S*. *S* ∈ *C* = (∀ *S'*. *finite S'* ⟶ *S'* ⊆ *S* ⟶ *S'* ∈ *C*))›

definition *mk-finite-char* :: ‹'*a set set* ⇒ '*a set set*› **where**
‹*mk-finite-char C* = {*S*. ∀ *S'*. *S'* ⊆ *S* ⟶ *finite S'* ⟶ *S'* ∈ *C*}›

theorem *finite-alt-consistency*:
assumes *altconc*: ‹*alt-consistency C*›
and ‹*subset-closed C*›
shows ‹*alt-consistency* (*mk-finite-char C*)›
unfolding *alt-consistency-def*
proof (*intro allI impI conjI*)
fix *S*
assume ‹*S* ∈ *mk-finite-char C*›
then have *finc*: ‹∀ *S'* ⊆ *S*. *finite S'* ⟶ *S'* ∈ *C*›
unfolding *mk-finite-char-def* **by** *blast*

have ‹∀ *S'* ∈ *C*. ∀ *S* ⊆ *S'*. *S* ∈ *C*›
using ‹*subset-closed C*› **unfolding** *subset-closed-def* **by** *blast*
then have *sc*: ‹∀ *S' x*. *S'* ∪ *x* ∈ *C* ⟶ (∀ *S* ⊆ *S'* ∪ *x*. *S* ∈ *C*)›
by *blast*

{ fix *p ts*
show ‹¬ (*Pred p ts* ∈ *S* ∧ *Neg* (*Pred p ts*) ∈ *S*)›
proof

```
    assume ‹Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S›
    then have ‹{Pred p ts, Neg (Pred p ts)} ∈ C›
      using finc by simp
    then show False
      using altconc unfolding alt-consistency-def by fast
  qed }

show ‹FF ∉ S›
proof
  assume ‹FF ∈ S›
  then have ‹{FF} ∈ C›
    using finc by simp
  then show False
    using altconc unfolding alt-consistency-def by fast
qed

show ‹Neg TT ∉ S›
proof
  assume ‹Neg TT ∈ S›
  then have ‹{Neg TT} ∈ C›
    using finc by simp
  then show False
    using altconc unfolding alt-consistency-def by fast
qed

{ fix Z
  assume ∗: ‹Neg (Neg Z) ∈ S›
  show ‹S ∪ {Z} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {Z} ∪ {Neg (Neg Z)}›

    assume ‹S' ⊆ S ∪ {Z}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Z} ∈ C›
      using altconc unfolding alt-consistency-def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix A B
  assume ∗: ‹And A B ∈ S›
  show ‹S ∪ {A, B} ∈ mk-finite-char C›
```

```
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {A, B} ∪ {And A B}›

    assume ‹S' ⊆ S ∪ {A, B}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {A, B} ∈ C›
      using altconc unfolding alt-consistency-def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix A B
  assume ∗: ‹Neg (Or A B) ∈ S›
  show ‹S ∪ {Neg A, Neg B} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {Neg A, Neg B} ∪ {Neg (Or A B)}›

    assume ‹S' ⊆ S ∪ {Neg A, Neg B}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg A, Neg B} ∈ C›
      using altconc unfolding alt-consistency-def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix A B
  assume ∗: ‹Neg (Impl A B) ∈ S›
  show ‹S ∪ {A, Neg B} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {A, Neg B} ∪ {Neg (Impl A B)}›

    assume ‹S' ⊆ S ∪ {A, Neg B}› and ‹finite S'›
    then have ‹?S' ⊆ S›
```

```
    using ∗ by blast
  moreover have ‹finite ?S'›
    using ‹finite S'› by blast
  ultimately have ‹?S' ∈ C›
    using finc by blast
  then have ‹?S' ∪ {A, Neg B} ∈ C›
    using altconc unfolding alt-consistency-def by simp
  then show ‹S' ∈ C›
    using sc by blast
 qed }

{ fix A B
 assume ∗: ‹Or A B ∈ S›
 show ‹S ∪ {A} ∈ mk-finite-char C ∨ S ∪ {B} ∈ mk-finite-char C›
 proof (rule ccontr)
  assume ‹¬ ?thesis›
  then obtain Sa and Sb
    where ‹Sa ⊆ S ∪ {A}› and ‹finite Sa› and ‹Sa ∉ C›
      and ‹Sb ⊆ S ∪ {B}› and ‹finite Sb› and ‹Sb ∉ C›
    unfolding mk-finite-char-def by blast

  let ?S' = ‹(Sa − {A}) ∪ (Sb − {B}) ∪ {Or A B}›

  have ‹?S' ⊆ S›
    using ‹Sa ⊆ S ∪ {A}› ‹Sb ⊆ S ∪ {B}› ∗ by blast
  moreover have ‹finite ?S'›
    using ‹finite Sa› ‹finite Sb› by blast
  ultimately have ‹?S' ∈ C›
    using finc by blast
  then have ‹?S' ∪ {A} ∈ C ∨ ?S' ∪ {B} ∈ C›
    using altconc unfolding alt-consistency-def by simp
  then have ‹Sa ∈ C ∨ Sb ∈ C›
    using sc by blast
  then show False
    using ‹Sa ∉ C› ‹Sb ∉ C› by blast
 qed }

{ fix A B
 assume ∗: ‹Neg (And A B) ∈ S›
 show ‹S ∪ {Neg A} ∈ mk-finite-char C ∨ S ∪ {Neg B} ∈ mk-finite-char C›
 proof (rule ccontr)
  assume ‹¬ ?thesis›
  then obtain Sa and Sb
    where ‹Sa ⊆ S ∪ {Neg A}› and ‹finite Sa› and ‹Sa ∉ C›
      and ‹Sb ⊆ S ∪ {Neg B}› and ‹finite Sb› and ‹Sb ∉ C›
    unfolding mk-finite-char-def by blast

  let ?S' = ‹(Sa − {Neg A}) ∪ (Sb − {Neg B}) ∪ {Neg (And A B)}›
```

```
    have ‹?S' ⊆ S›
      using ‹Sa ⊆ S ∪ {Neg A}› ‹Sb ⊆ S ∪ {Neg B}› ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite Sa› ‹finite Sb› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg A} ∈ C ∨ ?S' ∪ {Neg B} ∈ C›
      using altconc unfolding alt-consistency-def by simp
    then have ‹Sa ∈ C ∨ Sb ∈ C›
      using sc by blast
    then show False
      using ‹Sa ∉ C› ‹Sb ∉ C› by blast
  qed }

{ fix A B
  assume ∗: ‹Impl A B ∈ S›
  show ‹S ∪ {Neg A} ∈ mk-finite-char C ∨ S ∪ {B} ∈ mk-finite-char C›
  proof (rule ccontr)
    assume ‹¬ ?thesis›
    then obtain Sa and Sb
      where ‹Sa ⊆ S ∪ {Neg A}› and ‹finite Sa› and ‹Sa ∉ C›
        and ‹Sb ⊆ S ∪ {B}› and ‹finite Sb› and ‹Sb ∉ C›
      unfolding mk-finite-char-def by blast

    let ?S' = ‹(Sa − {Neg A}) ∪ (Sb − {B}) ∪ {Impl A B}›

    have ‹?S' ⊆ S›
      using ‹Sa ⊆ S ∪ {Neg A}› ‹Sb ⊆ S ∪ {B}› ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite Sa› ‹finite Sb› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg A} ∈ C ∨ ?S' ∪ {B} ∈ C›
      using altconc unfolding alt-consistency-def by simp
    then have ‹Sa ∈ C ∨ Sb ∈ C›
      using sc by blast
    then show False
      using ‹Sa ∉ C› ‹Sb ∉ C› by blast
  qed }

{ fix P and t :: ‹'a term›
  assume ∗: ‹Forall P ∈ S› and ‹closedt 0 t›
  show ‹S ∪ {P[t/0]} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {P[t/0]} ∪ {Forall P}›

    assume ‹S' ⊆ S ∪ {P[t/0]}› and ‹finite S'›
```

```
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {P[t/0]} ∈ C›
      using altconc ‹closedt 0 t› unfolding alt-consistency-def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix P and t :: ‹'a term›
  assume ∗: ‹Neg (Exists P) ∈ S› and ‹closedt 0 t›
  show ‹S ∪ {Neg (P[t/0])} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {Neg (P[t/0])} ∪ {Neg (Exists P)}›

    assume ‹S' ⊆ S ∪ {Neg (P[t/0])}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    then have ‹?S' ∪ {Neg (P[t/0])} ∈ C›
      using altconc ‹closedt 0 t› unfolding alt-consistency-def by simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }

{ fix P x
  assume ∗: ‹Exists P ∈ S› and ‹∀ a ∈ S. x ∉ params a›
  show ‹S ∪ {P[App x []/0]} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {P[App x []/0]} ∪ {Exists P}›

    assume ‹S' ⊆ S ∪ {P[App x []/0]}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    moreover have ‹∀ a ∈ ?S'. x ∉ params a›
```

```
      using ‹∀ a ∈ S. x ∉ params a› ‹?S' ⊆ S› by blast
    ultimately have ‹?S' ∪ {P[App x []/0]} ∈ C›
      using altconc ‹∀ a ∈ S. x ∉ params a› unfolding alt-consistency-def by
blast
    then show ‹S' ∈ C›
      using sc by blast
  qed }

 { fix P x
  assume ∗: ‹Neg (Forall P) ∈ S› and ‹∀ a ∈ S. x ∉ params a›
  show ‹S ∪ {Neg (P[App x []/0])} ∈ mk-finite-char C›
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let ?S' = ‹S' − {Neg (P[App x []/0])} ∪ {Neg (Forall P)}›

    assume ‹S' ⊆ S ∪ {Neg (P[App x []/0])}› and ‹finite S'›
    then have ‹?S' ⊆ S›
      using ∗ by blast
    moreover have ‹finite ?S'›
      using ‹finite S'› by blast
    ultimately have ‹?S' ∈ C›
      using finc by blast
    moreover have ‹∀ a ∈ ?S'. x ∉ params a›
      using ‹∀ a ∈ S. x ∉ params a› ‹?S' ⊆ S› by blast
    ultimately have ‹?S' ∪ {Neg (P[App x []/0])} ∈ C›
      using altconc ‹∀ a ∈ S. x ∉ params a› unfolding alt-consistency-def by
simp
    then show ‹S' ∈ C›
      using sc by blast
  qed }
qed

theorem finite-char: ‹finite-char (mk-finite-char C)›
  unfolding finite-char-def mk-finite-char-def by blast

theorem finite-char-closed: ‹finite-char C ⟹ subset-closed C›
  unfolding finite-char-def subset-closed-def
proof (intro ballI allI impI)
  fix S S'
  assume ∗: ‹∀ S. (S ∈ C) = (∀ S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C)›
    and ‹S' ∈ C› and ‹S ⊆ S'›
  then have ‹∀ S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C› by blast
  then show ‹S ∈ C› using ∗ by blast
qed

theorem finite-char-subset: ‹subset-closed C ⟹ C ⊆ mk-finite-char C›
  unfolding mk-finite-char-def subset-closed-def by blast
```

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*
  by *countable-datatype*

instance *form* :: (*countable*, *countable*) *countable*
  by *countable-datatype*

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*)›
  by (*induct n*) (*auto simp*: *is-chain-def*)

theorem *is-chainD'*:
  assumes ‹*is-chain f*› **and** ‹*x* ∈ *f m*› **and** ‹*m* ≤ *k*›
  shows ‹*x* ∈ *f k*›
proof −
  have ‹∃ *n*. *k* = *m* + *n*›
    using ‹*m* ≤ *k*› **by** (*simp add*: *le-iff-add*)
  then obtain *n* **where** ‹*k* = *m* + *n*›
    by *blast*
  then show ‹*x* ∈ *f k*›
    using ‹*is-chain f*› ‹*x* ∈ *f m*›
    by (*simp add*: *is-chainD*)
qed

theorem *chain-index*:
  assumes *ch*: ‹*is-chain f*› **and** *fin*: ‹*finite F*›
  shows ‹*F* ⊆ (⋃ *n*. *f n*) ⟹ ∃ *n*. *F* ⊆ *f n*›
  using *fin*
proof (*induct rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *blast*
next
  case (*insert x F*)
  then have ‹∃ *n*. *F* ⊆ *f n*› **and** ‹∃ *m*. *x* ∈ *f m*› **and** ‹*F* ⊆ (⋃ *x*. *f x*)›
    using *ch* **by** *simp-all*
  then obtain *n* **and** *m* **where** ‹*F* ⊆ *f n*› **and** ‹*x* ∈ *f m*›
    by *blast*

have ‹$m \leq max\ n\ m$› **and** ‹$n \leq max\ n\ m$›
by *simp-all*
have ‹$x \in f\ (max\ n\ m)$›
using *is-chainD'* *ch* ‹$x \in f\ m$› ‹$m \leq max\ n\ m$› **by** *fast*
moreover have ‹$F \subseteq f\ (max\ n\ m)$›
using *is-chainD'* *ch* ‹$F \subseteq f\ n$› ‹$n \leq max\ n\ m$› **by** *fast*
moreover have ‹$x \in f\ (max\ n\ m) \wedge F \subseteq f\ (max\ n\ m)$›
using *calculation* **by** *blast*
ultimately show *?case* **by** *blast*
qed

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 −
note ‹*finite S'*› **and** ‹$S' \subseteq (\bigcup n.\ f\ n)$›
then obtain *n* **where** ‹$S' \subseteq f\ n$›
using *chain-index* ‹*is-chain f*› **by** *blast*
moreover have ‹$f\ n \in C$›
using ‹$\forall n.\ f\ n \in C$› **by** *blast*
ultimately show ‹$S' \in C$›
using ‹$\forall S' \in C.\ \forall S \subseteq S'.\ S \in C$› **by** *blast*
qed

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 −
have ‹*subset-closed C*›
using *finite-char-closed* ‹*finite-char C*› **by** *blast*
then have ‹$\forall S' \in C.\ \forall S \subseteq S'.\ S \in C$›
using *subset-closed-def* **by** *blast*
then have ‹$\forall S'.\ finite\ S' \longrightarrow S' \subseteq (\bigcup n.\ f\ n) \longrightarrow S' \in C$›
using *chain-union-closed' assms* **by** *blast*
moreover have ‹$((\bigcup n.\ f\ n) \in C) = (\forall S'.\ finite\ S' \longrightarrow S' \subseteq (\bigcup n.\ f\ n) \longrightarrow S' \in C)$›
using ‹*finite-char C*› **unfolding** *finite-char-def* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

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

theorem finite-paramst [simp]: ‹finite (paramst (t :: 'a term))›
  ‹finite (paramsts (ts :: 'a term list))›
 by (induct t and ts rule: paramst.induct paramsts.induct) (simp-all split: sum.split)

theorem finite-params [simp]: ‹finite (params p)›
  by (induct p) simp-all

theorem finite-params-extend [simp]:
  ‹infinite (⋂ p ∈ S. - params p) ⟹ infinite (⋂ p ∈ extend S C f n. - params p)›
  by (induct n) simp-all

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

  have ‹infinite (- (⋃ x ∈ ?S'. params x))›
    using assms by simp
  then obtain x where ‹x ∈ - (⋃ x ∈ ?S'. params x)›
    using infinite-imp-nonempty by blast
  then have ‹∀ a ∈ ?S'. x ∉ params a›
    by blast
  then show ?thesis
```

by *blast*
qed

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 −
obtain *p* **where** ∗: ‹*f n* = *Exists p*›
using ‹∃ *p*. *f n* = *Exists p*› **by** *blast*
have ‹∃ *x*. *x* ∉ (⋃ *p* ∈ *?S′*. *params p*)›
using ‹*infinite* (− (⋃ *p* ∈ *S*. *params p*))› *infinite-params-available*
by *blast*
moreover have ‹*Exists p* ∈ *?S′*›
using ∗ **by** *simp*
then have ‹∀ *x*. *x* ∉ (⋃ *p* ∈ *?S′*. *params p*) ⟶ *?S′* ∪ {*p*[*App x* []/*0*]} ∈ *C*›
using ‹*?S′* ∈ *C*› ‹*alt-consistency C*›
unfolding *alt-consistency-def* **by** *simp*
ultimately have ‹(*?S′* ∪ {*p*[*App* (*SOME k*. *k* ∉ (⋃ *p* ∈ *?S′*. *params p*)) []/*0*]})
∈ *C*›
by (*metis* (*mono-tags*, *lifting*) *someI2*)
then show *?thesis*
using *assms* ∗ **by** *simp*
qed

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 −
obtain *p* **where** ∗: ‹*f n* = *Neg* (*Forall p*)›
using ‹∃ *p*. *f n* = *Neg* (*Forall p*)› **by** *blast*
have ‹∃ *x*. *x* ∉ (⋃ *p* ∈ *?S′*. *params p*)›
using ‹*infinite* (− (⋃ *p* ∈ *S*. *params p*))› *infinite-params-available*
by *blast*
moreover have ‹*Neg* (*Forall p*) ∈ *?S′*›
using ∗ **by** *simp*
then have ‹∀ *x*. *x* ∉ (⋃ *p* ∈ *?S′*. *params p*) ⟶ *?S′* ∪ {*Neg* (*p*[*App x* []/*0*])} ∈
C›
using ‹*?S′* ∈ *C*› ‹*alt-consistency C*›
unfolding *alt-consistency-def* **by** *simp*
ultimately have ‹(*?S′* ∪ {*Neg* (*p*[*App* (*SOME k*. *k* ∉ (⋃ *p* ∈ *?S′*. *params p*))
[]/*0*])}) ∈ *C*›
by (*metis* (*mono-tags*, *lifting*) *someI2*)

then show *?thesis*
using *assms* ∗ **by** *simp*
qed

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*›
using *assms* **by** *simp*

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*›
using *assms* **by** *simp*

theorem *extend-in-C*: ‹*alt-consistency C* ⟹
S ∈ *C* ⟹ *infinite* (− (⋃ *p* ∈ *S*. *params p*)) ⟹ *extend S C f n* ∈ *C*›
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
using *extend-in-C-Exists extend-in-C-Neg-Forall*
extend-in-C-no-delta extend-in-C-stop
by *metis*
qed

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*›
unfolding *Extend-def*
using *chain-union-closed is-chain-extend extend-in-C*
by *blast*

theorem *Extend-subset*: ‹*S* ⊆ *Extend S C f*›
proof
fix *x*
assume ‹*x* ∈ *S*›
then have ‹*x* ∈ *extend S C f* 0› **by** *simp*
then have ‹∃ *n*. *x* ∈ *extend S C f n*› **by** *blast*
then show ‹*x* ∈ *Extend S C f*› **by** (*simp add*: *Extend-def*)
qed

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*›
unfolding *maximal-def Extend-def*
proof (*intro ballI impI*)
fix *S'*
assume ‹*S'* ∈ *C*›
and ‹(⋃ *x*. *extend S C f x*) ⊆ *S'*›
moreover have ‹*S'* ⊆ (⋃ *x*. *extend S C f x*)›
proof (*rule ccontr*)
assume ‹¬ *S'* ⊆ (⋃ *x*. *extend S C f x*)›
then have ‹∃ *z*. *z* ∈ *S'* ∧ *z* ∉ (⋃ *x*. *extend S C f x*)›
by *blast*
then obtain *z* **where** ‹*z* ∈ *S'*› **and** ∗: ‹*z* ∉ (⋃ *x*. *extend S C f x*)›
by *blast*
then obtain *n* **where** ‹*z* = *f n*›
using ‹∀ *y*. ∃ *n*. *y* = *f n*› **by** *blast*

from ‹(⋃ *x*. *extend S C f x*) ⊆ *S'*› ‹*z* = *f n*› ‹*z* ∈ *S'*›
have ‹*extend S C f n* ∪ {*f n*} ⊆ *S'*› **by** *blast*

from ‹*finite-char C*›
have ‹*subset-closed C*› **using** *finite-char-closed* **by** *blast*
then have ‹∀ *S'* ∈ *C*. ∀ *S* ⊆ *S'*. *S* ∈ *C*›
unfolding *subset-closed-def* **by** *simp*
then have ‹∀ *S* ⊆ *S'*. *S* ∈ *C*›
using ‹*S'* ∈ *C*› **by** *blast*
then have ‹*extend S C f n* ∪ {*f n*} ∈ *C*›
using ‹*extend S C f n* ∪ {*f n*} ⊆ *S'*›
by *blast*
then have ‹*z* ∈ *extend S C f* (*Suc n*)›
using ‹*z* ∉ (⋃ *x*. *extend S C f x*)› ‹*z* = *f n*›
by *simp*
then show *False* **using** ∗ **by** *blast*
qed
ultimately show ‹(⋃ *x*. *extend S C f x*) = *S'*›
by *simp*
qed

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*)) ∧

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

In Herbrand models, each *closed* term is interpreted by itself. We introduce a new datatype *hterm* ("Herbrand terms"), which is similar to the datatype *term* introduced in §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*›
by (*induct t* **and** *ts rule*: *closedt.induct closedts.induct*) *simp-all*

theorem *herbrand-evalt'* [*simp*]:
‹*evalt e HApp (term-of-hterm ht) = ht*›
‹*evalts e HApp (terms-of-hterms hts) = hts*›
by (*induct ht* **and** *hts rule*: *term-of-hterm.induct terms-of-hterms.induct*) *simp-all*

theorem *closed-hterm* [*simp*]:
‹*closedt 0 (term-of-hterm (ht::'a hterm))*›
‹*closedts 0 (terms-of-hterms (hts::'a hterm list))*›
by (*induct ht* **and** *hts rule*: *term-of-hterm.induct terms-of-hterms.induct*) *simp-all*

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 (λa ts. Pred a (terms-of-hterms ts) ∈ H) (Neg p))›
proof (induct p rule: wf-induct [where r=‹measure size-form›])
  show ‹wf (measure size-form)›
    by blast
next
  let ?eval = ‹eval e HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H)›

  fix x
  assume wf: ‹∀ y. (y, x) ∈ measure size-form ⟶
                (y ∈ H ⟶ closed 0 y ⟶ ?eval y) ∧
              (Neg y ∈ H ⟶ closed 0 y ⟶ ?eval (Neg y))›

  show ‹(x ∈ H ⟶ closed 0 x ⟶ ?eval x) ∧ (Neg x ∈ H ⟶ closed 0 x ⟶
?eval (Neg x))›
  proof (cases x)
    case FF
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ H›
      then show ‹?eval x›
        using FF hin by (simp add: hintikka-def)
    next
      assume ‹Neg x ∈ H›
      then show ‹?eval (Neg x)› using FF by simp
    qed
  next
    case TT
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ H›
      then show ‹?eval x›
        using TT by simp
    next
      assume ‹Neg x ∈ H›
      then show ‹?eval (Neg x)›
        using TT hin by (simp add: hintikka-def)
    qed
  next
    case (Pred p ts)
    show ?thesis
    proof (intro conjI impI)
      assume ‹x ∈ H› and ‹closed 0 x›
      then show ‹?eval x› using Pred by simp
    next
      assume ‹Neg x ∈ H› and ‹closed 0 x›
```

```
    then have ‹Neg (Pred p ts) ∈ H›
      using Pred by simp
    then have ‹Pred p ts ∉ H›
      using hin unfolding hintikka-def by fast
    then show ‹?eval (Neg x)›
      using Pred ‹closed 0 x› by simp
  qed
next
  case (Neg Z)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then show ‹?eval x›
      using Neg wf by simp
  next
    assume ‹Neg x ∈ H›
    then have ‹Z ∈ H›
      using Neg hin unfolding hintikka-def by blast
    moreover assume ‹closed 0 x›
    then have ‹closed 0 Z›
      using Neg by simp
    ultimately have ‹?eval Z›
      using Neg wf by simp
    then show ‹?eval (Neg x)›
      using Neg by simp
  qed
next
  case (And A B)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹And A B ∈ H› and ‹closed 0 (And A B)›
      using And by simp-all
    then have ‹A ∈ H ∧ B ∈ H›
      using And hin unfolding hintikka-def by blast
    then show ‹?eval x›
      using And wf ‹closed 0 (And A B)› by simp
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (And A B) ∈ H› and ‹closed 0 (And A B)›
      using And by simp-all
    then have ‹Neg A ∈ H ∨ Neg B ∈ H›
      using hin unfolding hintikka-def by blast
    then show ‹?eval (Neg x)›
      using And wf ‹closed 0 (And A B)› by fastforce
  qed
next
  case (Or A B)
  then show ?thesis
```

```
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹Or A B ∈ H› and ‹closed 0 (Or A B)›
      using Or by simp-all
    then have ‹A ∈ H ∨ B ∈ H›
      using hin unfolding hintikka-def by blast
    then show ‹?eval x›
      using Or wf ‹closed 0 (Or A B)› by fastforce
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Or A B) ∈ H› and ‹closed 0 (Or A B)›
      using Or by simp-all
    then have ‹Neg A ∈ H ∧ Neg B ∈ H›
      using hin unfolding hintikka-def by blast
    then show ‹?eval (Neg x)›
      using Or wf ‹closed 0 (Or A B)› by simp
  qed
next
  case (Impl A B)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹Impl A B ∈ H› and ‹closed 0 (Impl A B)›
      using Impl by simp-all
    then have ‹Neg A ∈ H ∨ B ∈ H›
      using hin unfolding hintikka-def by blast
    then show ‹?eval x›
      using Impl wf ‹closed 0 (Impl A B)› by fastforce
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Impl A B) ∈ H› and ‹closed 0 (Impl A B)›
      using Impl by simp-all
    then have ‹A ∈ H ∧ Neg B ∈ H›
      using hin unfolding hintikka-def by blast
    then show ‹?eval (Neg x)›
      using Impl wf ‹closed 0 (Impl A B)› by simp
  qed
next
  case (Forall P)
  then show ?thesis
  proof (intro conjI impI)
    assume ‹x ∈ H› and ‹closed 0 x›
    have ‹∀ z. eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H) P›
    proof (rule allI)
      fix z
      from ‹x ∈ H› and ‹closed 0 x›
      have ‹Forall P ∈ H› and ‹closed 0 (Forall P)›
        using Forall by simp-all
      then have ∗: ‹∀ P t. closedt 0 t ⟶ Forall P ∈ H ⟶ P[t/0] ∈ H›
```

```
        using hin unfolding hintikka-def by blast
      from ‹closed 0 (Forall P)›
      have ‹closed (Suc 0) P› by simp

      have ‹(P[term-of-hterm z/0], Forall P) ∈ measure size-form ⟶
          (P[term-of-hterm z/0] ∈ H ⟶ closed 0 (P[term-of-hterm z/0]) ⟶
          ?eval (P[term-of-hterm z/0]))›
        using Forall wf by blast
      then show ‹eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H)
P›
        using ∗ ‹Forall P ∈ H› ‹closed (Suc 0) P› by simp
    qed
    then show ‹?eval x›
      using Forall by simp
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    then have ‹Neg (Forall P) ∈ H›
      using Forall by simp
    then have ‹∃ t. closedt 0 t ∧ Neg (P[t/0]) ∈ H›
      using Forall hin unfolding hintikka-def by blast
    then obtain t where ∗: ‹closedt 0 t ∧ Neg (P[t/0]) ∈ H›
      by blast
    then have ‹closed 0 (P[t/0])›
      using Forall ‹closed 0 x› by simp

    have ‹(subst P t 0, Forall P) ∈ measure size-form ⟶
          (Neg (subst P t 0) ∈ H ⟶ closed 0 (subst P t 0) ⟶
          ?eval (Neg (subst P t 0)))›
      using Forall wf by blast
    then have ‹?eval (Neg (P[t/0]))›
      using Forall ∗ ‹closed 0 (P[t/0])› by simp
    then have ‹∃ z. ¬ eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈
H) P›
      by auto
    then show ‹?eval (Neg x)›
      using Forall by simp
  qed
next
  case (Exists P)
  then show ?thesis
  proof (intro conjI impI allI)
    assume ‹x ∈ H› and ‹closed 0 x›
    then have ‹∃ t. closedt 0 t ∧ (P[t/0]) ∈ H›
      using Exists hin unfolding hintikka-def by blast
    then obtain t where ∗: ‹closedt 0 t ∧ (P[t/0]) ∈ H›
      by blast
    then have ‹closed 0 (P[t/0])›
      using Exists ‹closed 0 x› by simp
```

```
    have ‹(subst P t 0, Exists P) ∈ measure size-form ⟶
          ((subst P t 0) ∈ H ⟶ closed 0 (subst P t 0) ⟶
          ?eval (subst P t 0))›
      using Exists wf by blast
    then have ‹?eval (P[t/0])›
      using Exists ∗ ‹closed 0 (P[t/0])› by simp
   then have ‹∃ z. eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H)
P›
      by auto
    then show ‹?eval x›
      using Exists by simp
  next
    assume ‹Neg x ∈ H› and ‹closed 0 x›
    have ‹∀ z. ¬ eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H) P›
    proof (rule allI)
      fix z
      from ‹Neg x ∈ H› and ‹closed 0 x›
      have ‹Neg (Exists P) ∈ H› and ‹closed 0 (Neg (Exists P))›
        using Exists by simp-all
      then have ∗: ‹∀ P t. closedt 0 t ⟶ Neg (Exists P) ∈ H ⟶ Neg (P[t/0])
∈ H›
        using hin unfolding hintikka-def by blast
      from ‹closed 0 (Neg (Exists P))›
      have ‹closed (Suc 0) P› by simp

      have ‹(P[term-of-hterm z/0], Exists P) ∈ measure size-form ⟶
          (Neg (P[term-of-hterm z/0]) ∈ H ⟶ closed 0 (P[term-of-hterm z/0])
⟶
          ?eval (Neg (P[term-of-hterm z/0])))›
        using Exists wf by blast
     then show ‹¬ eval (e⟨0:z⟩) HApp (λa ts. Pred a (terms-of-hterms ts) ∈ H)
P›
        using ∗ ‹Neg (Exists P) ∈ H› ‹closed (Suc 0) P› by simp
    qed
    then show ‹?eval (Neg x)›
      using Exists by simp
  qed
 qed
qed
```

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 −
have ‹∃ *p*. *f n* = *Exists p*›
using ‹*Exists P* = *f n*› **by** *metis*
then have ‹*extend S C f* (*Suc n*) = (*?S′* ∪ {(*dest-Exists* (*f n*))[*?t*/0]})›
using ‹*?S′* ∈ *C*› **by** *simp*
also have ‹. . . = (*?S′* ∪ {(*dest-Exists* (*Exists P*))[*?t*/0]})›
using ‹*Exists P* = *f n*› **by** *simp*
also have ‹. . . = (*?S′* ∪ {*P*[*?t*/0]})›
by *simp*
finally show *?thesis*
by *blast*
qed

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 −
have ‹*f n* ≠ *Exists P*›
using ‹*Neg* (*Forall P*) = *f n*› **by** *auto*

have ‹∃ *p*. *f n* = *Neg* (*Forall p*)›
using ‹*Neg* (*Forall P*) = *f n*› **by** *metis*
then have ‹*extend S C f* (*Suc n*) = (*?S′* ∪ {*Neg* (*dest-Forall* (*dest-Neg* (*f n*))[*?t*/0])})›
using ‹*?S′* ∈ *C*› ‹*f n* ≠ *Exists P*› **by** *auto*
also have ‹. . . = (*?S′* ∪ {*Neg* (*dest-Forall* (*dest-Neg* (*Neg* (*Forall P*)))[*?t*/0])})›
using ‹*Neg* (*Forall P*) = *f n*› **by** *simp*
also have ‹. . . = (*?S′* ∪ {*Neg* (*P*[*?t*/0])})›
by *simp*
finally show *?thesis*
by *blast*
qed

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*›)
unfolding *hintikka-def*
proof (*intro allI impI conjI*)
have ‹*maximal ?H C*›
by (*simp add*: *extend-maximal fin-ch surj*)

have ‹*?H* ∈ *C*›
using *Extend-in-C assms* **by** *blast*

have ‹∀ *S'* ∈ *C*. *?H* ⊆ *S'* ⟶ *?H* = *S'*›
using ‹*maximal ?H C*›
unfolding *maximal-def* **by** *blast*

{ fix *p ts*
show ‹¬ (*Pred p ts* ∈ *?H* ∧ *Neg* (*Pred p ts*) ∈ *?H*)›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *fast* **}**

show ‹*FF* ∉ *?H*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *blast*

show ‹*Neg TT* ∉ *?H*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *blast*

{ fix *Z*
assume ‹*Neg* (*Neg Z*) ∈ *?H*›
then have ‹*?H* ∪ {*Z*} ∈ *C*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *fast*
then show ‹*Z* ∈ *?H*›
using ‹*maximal ?H C*› **unfolding** *maximal-def* **by** *fast* **}**

{ fix *A B*
assume ‹*And A B* ∈ *?H*›
then have ‹*?H* ∪ {*A*, *B*} ∈ *C*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *fast*
then show ‹*A* ∈ *?H*› **and** ‹*B* ∈ *?H*›
using ‹*maximal ?H C*› **unfolding** *maximal-def* **by** *fast+* **}**

{ fix *A B*
assume ‹*Neg* (*Or A B*) ∈ *?H*›
then have ‹*?H* ∪ {*Neg A*, *Neg B*} ∈ *C*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *fast*
then show ‹*Neg A* ∈ *?H*› **and** ‹*Neg B* ∈ *?H*›
using ‹*maximal ?H C*› **unfolding** *maximal-def* **by** *fast+* **}**

{ fix *A B*
assume ‹*Neg* (*Impl A B*) ∈ *?H*›
then have ‹*?H* ∪ {*A*, *Neg B*} ∈ *C*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *blast*
then show ‹*A* ∈ *?H*› **and** ‹*Neg B* ∈ *?H*›
using ‹*maximal ?H C*› **unfolding** *maximal-def* **by** *fast+* **}**

{ fix *A B*
assume ‹*Or A B* ∈ *?H*›
then have ‹*?H* ∪ {*A*} ∈ *C* ∨ *?H* ∪ {*B*} ∈ *C*›
using ‹*?H* ∈ *C*› *altc* **unfolding** *alt-consistency-def* **by** *fast*

```
  then show ‹A ∈ ?H ∨ B ∈ ?H›
    using ‹maximal ?H C› unfolding maximal-def by fast }

{ fix A B
  assume ‹Neg (And A B) ∈ ?H›
  then have ‹?H ∪ {Neg A} ∈ C ∨ ?H ∪ {Neg B} ∈ C›
    using ‹?H ∈ C› altc unfolding alt-consistency-def by simp
  then show ‹Neg A ∈ ?H ∨ Neg B ∈ ?H›
    using ‹maximal ?H C› unfolding maximal-def by fast }

{ fix A B
  assume ‹Impl A B ∈ ?H›
  then have ‹?H ∪ {Neg A} ∈ C ∨ ?H ∪ {B} ∈ C›
    using ‹?H ∈ C› altc unfolding alt-consistency-def by simp
  then show ‹Neg A ∈ ?H ∨ B ∈ ?H›
    using ‹maximal ?H C› unfolding maximal-def by fast }

{ fix P and t :: ‹nat term›
  assume ‹Forall P ∈ ?H› and ‹closedt 0 t›
  then have ‹?H ∪ {P[t/0]} ∈ C›
    using ‹?H ∈ C› altc unfolding alt-consistency-def by blast
  then show ‹P[t/0] ∈ ?H›
    using ‹maximal ?H C› unfolding maximal-def by fast }

{ fix P and t :: ‹nat term›
  assume ‹Neg (Exists P) ∈ ?H› and ‹closedt 0 t›
  then have ‹?H ∪ {Neg (P[t/0])} ∈ C›
    using ‹?H ∈ C› altc unfolding alt-consistency-def by blast
  then show ‹Neg (P[t/0]) ∈ ?H›
    using ‹maximal ?H C› unfolding maximal-def by fast }

{ fix P
  assume ‹Exists P ∈ ?H›
  obtain n where *: ‹Exists P = f n›
    using surj by blast

  let ?t = ‹App (SOME k. k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) []›
  have ‹closedt 0 ?t› by simp

  have ‹Exists P ∈ (⋃n. extend S C f n)›
    using ‹Exists P ∈ ?H› Extend-def by blast
  then have ‹extend S C f n ∪ {f n} ⊆ (⋃n. extend S C f n)›
    using * by (simp add: UN-upper)
  then have ‹extend S C f n ∪ {f n} ∈ C›
    using Extend-def ‹Extend S C f ∈ C› fin-ch finite-char-closed
    unfolding subset-closed-def by metis
  then have ‹P[?t/0] ∈ extend S C f (Suc n)›
    using * Exists-in-extend by blast
  then have ‹P[?t/0] ∈ ?H›
```

```
    using Extend-def by blast
  then show ‹∃ t. closedt 0 t ∧ P[t/0] ∈ ?H›
    using ‹closedt 0 ?t› by blast }

 { fix P
  assume ‹Neg (Forall P) ∈ ?H›
  obtain n where *: ‹Neg (Forall P) = f n›
    using surj by blast

  let ?t = ‹App (SOME k. k ∉ (⋃p ∈ extend S C f n ∪ {f n}. params p)) []›
  have ‹closedt 0 ?t› by simp

  have ‹Neg (Forall P) ∈ (⋃n. extend S C f n)›
    using ‹Neg (Forall P) ∈ ?H› Extend-def by blast
  then have ‹extend S C f n ∪ {f n} ⊆ (⋃n. extend S C f n)›
    using * by (simp add: UN-upper)
  then have ‹extend S C f n ∪ {f n} ∈ C›
    using Extend-def ‹Extend S C f ∈ C› fin-ch finite-char-closed
    unfolding subset-closed-def by metis
  then have ‹Neg (P[?t/0]) ∈ extend S C f (Suc n)›
    using * Neg-Forall-in-extend by blast
  then have ‹Neg (P[?t/0]) ∈ ?H›
    using Extend-def by blast
  then show ‹∃ t. closedt 0 t ∧ Neg (P[t/0]) ∈ ?H›
    using ‹closedt 0 ?t› by blast }
qed
```

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 −
  have ‹finite-char ?C'›
    using finite-char by blast
  moreover have ‹∀ y. y = from-nat (to-nat y)›
    by simp
  then have ‹∀ y. ∃ n. y = from-nat n›
    by blast
  moreover have ‹alt-consistency ?C'›
    using alt-consistency close-closed close-consistency ‹consistency C›
      finite-alt-consistency mk-alt-consistency-closed
    by blast
  moreover have ‹S ∈ close C›
    using close-subset ‹S ∈ C› by blast
```

```
  then have ‹S ∈ mk-alt-consistency (close C)›
    using mk-alt-consistency-subset by blast
  then have ‹S ∈ ?C'›
    using close-closed finite-char-subset mk-alt-consistency-closed by blast
  ultimately show ?thesis
    using extend-hintikka ‹infinite (− (⋃p ∈ S. params p))›
    by metis
qed

theorem model-existence:
  assumes ‹consistency C›
    and ‹S ∈ C›
    and ‹infinite (− (⋃p ∈ S. params p))›
    and ‹p ∈ S›
    and ‹closed 0 p›
  shows ‹eval e HApp (λa ts. Pred a (terms-of-hterms ts) ∈ Extend S
        (mk-finite-char (mk-alt-consistency (close C))) from-nat) p›
  using assms hintikka-model hintikka-Extend-S Extend-subset
  by blast
```

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. ∃ G. S = set G ∧ ¬ G ⊢ FF}›
  unfolding consistency-def
proof (intro conjI allI impI notI)
  fix S :: ‹('a, 'b) form set›
  assume ‹S ∈ {set G | G. ¬ G ⊢ FF}› (is ‹S ∈ ?C›)
  then obtain G :: ‹('a, 'b) form list›
    where ∗: ‹S = set G› and ‹¬ G ⊢ FF›
    by blast

  { fix p ts
    assume ‹Pred p ts ∈ S ∧ Neg (Pred p ts) ∈ S›
    then have ‹G ⊢ Pred p ts› and ‹G ⊢ Neg (Pred p ts)›
      using Assum ∗ by blast+
    then have ‹G ⊢ FF›
      using NegE by blast
    then show False
      using ‹¬ G ⊢ FF› by blast }

  { assume ‹FF ∈ S›
    then have ‹G ⊢ FF›
```

```
    using Assum * by blast
  then show False
    using ‹¬ G ⊢ FF› by blast }

{ assume ‹Neg TT ∈ S›
  then have ‹G ⊢ Neg TT›
    using Assum * by blast
  moreover have ‹G ⊢ TT›
    using TTI by blast
  ultimately have ‹G ⊢ FF›
    using NegE by blast
  then show False
    using ‹¬ G ⊢ FF› by blast }

{ fix Z
  assume ‹Neg (Neg Z) ∈ S›
  then have ‹G ⊢ Neg (Neg Z)›
    using Assum * by blast

  { assume ‹Z # G ⊢ FF›
    then have ‹G ⊢ Neg Z›
      using NegI by blast
    then have ‹G ⊢ FF›
      using NegE ‹G ⊢ Neg (Neg Z)› by blast
    then have False
      using ‹¬ G ⊢ FF› by blast }
  then have ‹¬ Z # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {Z} = set (Z # G)›
    using * by simp
  ultimately show ‹S ∪ {Z} ∈ ?C›
    by blast }

{ fix A B
  assume ‹And A B ∈ S›
  then have ‹G ⊢ And A B›
    using Assum * by blast
  then have ‹G ⊢ A› and ‹G ⊢ B›
    using AndE1 AndE2 by blast+

  { assume ‹A # B # G ⊢ FF›
    then have ‹B # G ⊢ Neg A›
      using NegI by blast
    then have ‹G ⊢ Neg A›
      using cut ‹G ⊢ B› by blast
    then have ‹G ⊢ FF›
      using NegE ‹G ⊢ A› by blast
    then have False
      using ‹¬ G ⊢ FF› by blast }
```

```
  then have ‹¬ A # B # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {A, B} = set (A # B # G)›
    using ∗ by simp
  ultimately show ‹S ∪ {A, B} ∈ ?C›
    by blast }

{ fix A B
  assume ‹Neg (Or A B) ∈ S›
  then have ‹G ⊢ Neg (Or A B)›
    using Assum ∗ by blast

  have ‹A # Neg B # G ⊢ A›
    by (simp add: Assum)
  then have ‹A # Neg B # G ⊢ Or A B›
    using OrI1 by blast
  moreover have ‹A # Neg B # G ⊢ Neg (Or A B)›
    using ∗ ‹Neg (Or A B) ∈ S› by (simp add: Assum)
  ultimately have ‹A # Neg B # G ⊢ FF›
    using NegE ‹A # Neg B # G ⊢ Neg (Or A B)› by blast
  then have ‹Neg B # G ⊢ Neg A›
    using NegI by blast

  have ‹B # G ⊢ B›
    by (simp add: Assum)
  then have ‹B # G ⊢ Or A B›
    using OrI2 by blast
  moreover have ‹B # G ⊢ Neg (Or A B)›
    using ∗ ‹Neg (Or A B) ∈ S› by (simp add: Assum)
  ultimately have ‹B # G ⊢ FF›
    using NegE ‹B # G ⊢ Neg (Or A B)› by blast
  then have ‹G ⊢ Neg B›
    using NegI by blast

  { assume ‹Neg A # Neg B # G ⊢ FF›
    then have ‹Neg B # G ⊢ Neg (Neg A)›
      using NegI by blast
    then have ‹Neg B # G ⊢ FF›
      using NegE ‹Neg B # G ⊢ Neg A› by blast
    then have ‹G ⊢ FF›
      using cut ‹G ⊢ Neg B› by blast
    then have False
      using ‹¬ G ⊢ FF› by blast }
  then have ‹¬ Neg A # Neg B # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {Neg A, Neg B} = set (Neg A # Neg B # G)›
    using ∗ by simp
  ultimately show ‹S ∪ {Neg A, Neg B} ∈ ?C›
    by blast }
```

```
{ fix A B
  assume ‹Neg (Impl A B) ∈ S›

  have ‹A # Neg A # Neg B # G ⊢ A›
    by (simp add: Assum)
  moreover have ‹A # Neg A # Neg B # G ⊢ Neg A›
    by (simp add: Assum)
  ultimately have ‹A # Neg A # Neg B # G ⊢ FF›
    using NegE by blast
  then have ‹A # Neg A # Neg B # G ⊢ B›
    using FFE by blast
  then have ‹Neg A # Neg B # G ⊢ Impl A B›
    using ImplI by blast
  moreover have ‹Neg A # Neg B # G ⊢ Neg (Impl A B)›
    using ∗ ‹Neg (Impl A B) ∈ S› by (simp add: Assum)
  ultimately have ‹Neg A # Neg B # G ⊢ FF›
    using NegE by blast
  then have ‹Neg B # G ⊢ A›
    using Class by blast

  have ‹A # B # G ⊢ B›
    by (simp add: Assum)
  then have ‹B # G ⊢ Impl A B›
    using ImplI by blast
  moreover have ‹B # G ⊢ Neg (Impl A B)›
    using ∗ ‹Neg (Impl A B) ∈ S› by (simp add: Assum)
  ultimately have ‹B # G ⊢ FF›
    using NegE by blast
  then have ‹G ⊢ Neg B›
    using NegI by blast

  { assume ‹A # Neg B # G ⊢ FF›
    then have ‹Neg B # G ⊢ Neg A›
      using NegI by blast
    then have ‹Neg B # G ⊢ FF›
      using NegE ‹Neg B # G ⊢ A› by blast
    then have ‹G ⊢ FF›
      using cut ‹G ⊢ Neg B› by blast
    then have False
      using ‹¬ G ⊢ FF›
      by blast }
  then have ‹¬ A # Neg B # G ⊢ FF›
    by blast
  moreover have ‹{A, Neg B} ∪ S = set (A # Neg B # G)›
    using ∗ by simp
  ultimately show ‹S ∪ {A, Neg B} ∈ ?C›
    by blast }
```

{ fix A B
assume ‹$Or\ A\ B \in S$›
then have ‹$G \vdash Or\ A\ B$›
using $*$ *Assum* **by** *blast*

{ assume ‹$(\forall G'.\ set\ G' = S \cup \{A\} \longrightarrow G' \vdash FF)$›
and ‹$(\forall G'.\ set\ G' = S \cup \{B\} \longrightarrow G' \vdash FF)$›
then have ‹$A\ \#\ G \vdash FF$› **and** ‹$B\ \#\ G \vdash FF$›
using $*$ **by** *simp-all*
then have ‹$G \vdash FF$›
using *OrE* ‹$G \vdash Or\ A\ B$› **by** *blast*
then have *False*
using ‹$\neg\ G \vdash FF$› **by** *blast* **}**
then show ‹$S \cup \{A\} \in ?C \vee S \cup \{B\} \in ?C$›
by *blast* **}**

{ fix A B
assume ‹$Neg\ (And\ A\ B) \in S$›

let $?x$ = ‹$Or\ (Neg\ A)\ (Neg\ B)$›

have ‹$B\ \#\ A\ \#\ Neg\ ?x\ \#\ G \vdash A$› **and** ‹$B\ \#\ A\ \#\ Neg\ ?x\ \#\ G \vdash B$›
by (*simp-all add*: *Assum*)
then have ‹$B\ \#\ A\ \#\ Neg\ ?x\ \#\ G \vdash And\ A\ B$›
using *AndI* **by** *blast*
moreover have ‹$B\ \#\ A\ \#\ Neg\ ?x\ \#\ G \vdash Neg\ (And\ A\ B)$›
using $*$ ‹$Neg\ (And\ A\ B) \in S$› **by** (*simp add*: *Assum*)
ultimately have ‹$B\ \#\ A\ \#\ Neg\ ?x\ \#\ G \vdash FF$›
using *NegE* **by** *blast*
then have ‹$A\ \#\ Neg\ ?x\ \#\ G \vdash Neg\ B$›
using *NegI* **by** *blast*
then have ‹$A\ \#\ Neg\ ?x\ \#\ G \vdash ?x$›
using *OrI2* **by** *blast*
moreover have ‹$A\ \#\ Neg\ ?x\ \#\ G \vdash Neg\ ?x$›
by (*simp add*: *Assum*)
ultimately have ‹$A\ \#\ Neg\ ?x\ \#\ G \vdash FF$›
using *NegE* **by** *blast*
then have ‹$Neg\ ?x\ \#\ G \vdash Neg\ A$›
using *NegI* **by** *blast*
then have ‹$Neg\ ?x\ \#\ G \vdash ?x$›
using *OrI1* **by** *blast*
then have ‹$G \vdash Or\ (Neg\ A)\ (Neg\ B)$›
using *Class'* **by** *blast*

{ assume ‹$(\forall G'.\ set\ G' = S \cup \{Neg\ A\} \longrightarrow G' \vdash FF)$›
and ‹$(\forall G'.\ set\ G' = S \cup \{Neg\ B\} \longrightarrow G' \vdash FF)$›
then have ‹$Neg\ A\ \#\ G \vdash FF$› **and** ‹$Neg\ B\ \#\ G \vdash FF$›
using $*$ **by** *simp-all*
then have ‹$G \vdash FF$›

```
    using OrE ‹G ⊢ Or (Neg A) (Neg B)› by blast
   then have False
    using ‹¬ G ⊢ FF› by blast }
  then show ‹S ∪ {Neg A} ∈ ?C ∨ S ∪ {Neg B} ∈ ?C›
   by blast }

{ fix A B
  assume ‹Impl A B ∈ S›

  let ?x = ‹Or (Neg A) B›

  have ‹A # Neg ?x # G ⊢ A›
   by (simp add: Assum)
  moreover have ‹A # Neg ?x # G ⊢ Impl A B›
   using ∗ ‹Impl A B ∈ S› by (simp add: Assum)
  ultimately have ‹A # Neg ?x # G ⊢ B›
   using ImplE by blast
  then have ‹A # Neg ?x # G ⊢ ?x›
   using OrI2 by blast
  moreover have ‹A # Neg ?x # G ⊢ Neg ?x›
   by (simp add: Assum)
  ultimately have ‹A # Neg ?x # G ⊢ FF›
   using NegE by blast
  then have ‹Neg ?x # G ⊢ Neg A›
   using NegI by blast
  then have ‹Neg ?x # G ⊢ ?x›
   using OrI1 by blast
  then have ‹G ⊢ Or (Neg A) B›
   using Class′ by blast

  { assume ‹(∀ G′. set G′ = S ∪ {Neg A} ⟶ G′ ⊢ FF)›
     and ‹(∀ G′. set G′ = S ∪ {B} ⟶ G′ ⊢ FF)›
    then have ‹Neg A # G ⊢ FF› and ‹B # G ⊢ FF›
     using ∗ by simp-all
    then have ‹G ⊢ FF›
     using OrE ‹G ⊢ Or (Neg A) B› by blast
    then have False
     using ‹¬ G ⊢ FF› by blast }
  then show ‹S ∪ {Neg A} ∈ ?C ∨ S ∪ {B} ∈ ?C›
   by blast }

{ fix P and t :: ‹'a term›
  assume ‹closedt 0 t› and ‹Forall P ∈ S›
  then have ‹G ⊢ Forall P›
   using Assum ∗ by blast
  then have ‹G ⊢ P[t/0]›
   using ForallE by blast

  { assume ‹P[t/0] # G ⊢ FF›
```

```
    then have ‹G ⊢ FF›
      using cut ‹G ⊢ P[t/0]› by blast
    then have False
      using ‹¬ G ⊢ FF› by blast }
  then have ‹¬ P[t/0] # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {P[t/0]} = set (P[t/0] # G)›
    using ∗ by simp
  ultimately show ‹S ∪ {P[t/0]} ∈ ?C›
    by blast }

{ fix P and t :: ‹'a term›
  assume ‹closedt 0 t› and ‹Neg (Exists P) ∈ S›
  then have ‹G ⊢ Neg (Exists P)›
    using Assum ∗ by blast
  then have ‹P[t/0] ∈ set (P[t/0] # G)›
    by (simp add: Assum)
  then have ‹P[t/0] # G ⊢ P[t/0]›
    using Assum by blast
  then have ‹P[t/0] # G ⊢ Exists P›
    using ExistsI by blast
  moreover have ‹P[t/0] # G ⊢ Neg (Exists P)›
    using ∗ ‹Neg (Exists P) ∈ S› by (simp add: Assum)
  ultimately have ‹P[t/0] # G ⊢ FF›
    using NegE by blast
  then have ‹G ⊢ Neg (P[t/0])›
    using NegI by blast

  { assume ‹Neg (P[t/0]) # G ⊢ FF›
    then have ‹G ⊢ FF›
      using cut ‹G ⊢ Neg (P[t/0])› by blast
    then have False
      using ‹¬ G ⊢ FF› by blast }
  then have ‹¬ Neg (P[t/0]) # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {Neg (P[t/0])} = set (Neg (P[t/0]) # G)›
    using ∗ by simp
  ultimately show ‹S ∪ {Neg (P[t/0])} ∈ ?C›
    by blast }

{ fix P
  assume ‹Exists P ∈ S›
  then have ‹G ⊢ Exists P›
    using ∗ Assum by blast

  have ‹finite ((⋃p ∈ set G. params p) ∪ params P)›
    by simp
  then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
    using inf-param Diff-infinite-finite finite-compl by blast
```

```
  then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
    by (simp add: Compl-eq-Diff-UNIV)
  then obtain x where **: ‹x ∈ − ((⋃p ∈ set G. params p) ∪ params P)›
    using infinite-imp-nonempty by blast

  { assume ‹P[App x []/0] # G ⊢ FF›
    moreover have ‹list-all (λp. x ∉ params p) G›
      using ** by (simp add: list-all-iff)
    moreover have ‹x ∉ params P›
      using ** by simp
    moreover have ‹x ∉ params FF›
      by simp
    ultimately have ‹G ⊢ FF›
      using ExistsE ‹G ⊢ Exists P› by fast
    then have False
      using ‹¬ G ⊢ FF›
      by blast}
  then have ‹¬ P[App x []/0] # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {P[App x []/0]} = set (P[App x []/0] # G)›
    using * by simp
  ultimately show ‹∃x. S ∪ {P[App x []/0]} ∈ ?C›
    by blast }

{ fix P
  assume ‹Neg (Forall P) ∈ S›
  then have ‹G ⊢ Neg (Forall P)›
    using * Assum by blast

  have ‹finite ((⋃p ∈ set G. params p) ∪ params P)›
    by simp
  then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
    using inf-param Diff-infinite-finite finite-compl by blast
  then have ‹infinite (− ((⋃p ∈ set G. params p) ∪ params P))›
    by (simp add: Compl-eq-Diff-UNIV)
  then obtain x where **: ‹x ∈ − ((⋃p ∈ set G. params p) ∪ params P)›
    using infinite-imp-nonempty by blast

  let ?x = ‹Neg (Exists (Neg P))›

  have ‹Neg (P[App x []/0]) # ?x # G ⊢ Neg P[App x []/0]›
    by (simp add: Assum)
  then have ‹Neg (P[App x []/0]) # ?x # G ⊢ Exists (Neg P)›
    using ExistsI by blast
  moreover have ‹Neg (P[App x []/0]) # ?x # G ⊢ ?x›
    by (simp add: Assum)
  ultimately have ‹Neg (P[App x []/0]) # ?x # G ⊢ FF›
    using NegE by blast
  then have ‹?x # G ⊢ P[App x []/0]›
```

```
    using Class by blast
  moreover have ‹list-all (λp. x ∉ params p) (?x # G)›
    using ∗∗ by (simp add: list-all-iff)
  moreover have ‹x ∉ params P›
    using ∗∗ by simp
  ultimately have ‹?x # G ⊢ Forall P›
    using ForallI by fast
  moreover have ‹?x # G ⊢ Neg (Forall P)›
    using ∗ ‹Neg (Forall P) ∈ S› by (simp add: Assum)
  ultimately have ‹?x # G ⊢ FF›
    using NegE by blast
  then have ‹G ⊢ Exists (Neg P)›
    using Class by blast

  { assume ‹Neg (P[App x []/0]) # G ⊢ FF›
    moreover have ‹list-all (λp. x ∉ params p) G›
      using ∗∗ by (simp add: list-all-iff)
    moreover have ‹x ∉ params P›
      using ∗∗ by simp
    moreover have ‹x ∉ params FF›
      by simp
    ultimately have ‹G ⊢ FF›
      using ExistsE ‹G ⊢ Exists (Neg P)› by fastforce
    then have False
      using ‹¬ G ⊢ FF›
      by blast}
  then have ‹¬ Neg (P[App x []/0]) # G ⊢ FF›
    by blast
  moreover have ‹S ∪ {Neg (P[App x []/0])} = set (Neg (P[App x []/0]) # G)›
    using ∗ by simp
  ultimately show ‹∃x. S ∪ {Neg (P[App x []/0])} ∈ ?C›
    by blast }
qed
```

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

```
theorem natded-complete:
  assumes ‹closed 0 p›
    and ‹list-all (closed 0) ps›
    and mod: ‹∀ e f g. e,(f :: nat ⇒ nat hterm list ⇒ nat hterm),
            (g :: nat ⇒ nat hterm list ⇒ bool),ps ⊨ p›
  shows ‹ps ⊢ p›
proof (rule Class, rule ccontr)
  fix e
  assume ‹¬ Neg p # ps ⊢ FF›

  let ?S = ‹set (Neg p # ps)›
  let ?C = ‹{set (G :: (nat, nat) form list) | G. ¬ G ⊢ FF}›
  let ?f = HApp
  let ?g = ‹(λa ts. Pred a (terms-of-hterms ts) ∈ Extend ?S
```

(*mk-finite-char* (*mk-alt-consistency* (*close* *?C*))) *from-nat*)›

from ‹*list-all* (*closed* 0) *ps*›
have ‹∀ *p* ∈ *set ps*. *closed* 0 *p*›
by (*simp add*: *list-all-iff*)

{ fix *x*
assume ‹*x* ∈ *?S*›
moreover have ‹*consistency ?C*›
using *deriv-consistency* **by** *blast*
moreover have ‹*?S* ∈ *?C*›
using ‹¬ *Neg p* # *ps* ⊢ *FF*› **by** *blast*
moreover have ‹*infinite* (− (⋃ *p* ∈ *?S*. *params p*))›
by (*simp add*: *Compl-eq-Diff-UNIV*)
moreover note ‹*closed* 0 *p*› ‹∀ *p* ∈ *set ps*. *closed* 0 *p*› ‹*x* ∈ *?S*›
then have ‹*closed* 0 *x*› **by** *auto*
ultimately have ‹*eval e ?f ?g x*›
using *model-existence* **by** *blast* **}**
then have ‹*list-all* (*eval e ?f ?g*) (*Neg p* # *ps*)›
by (*simp add*: *list-all-iff*)
moreover have ‹*eval e ?f ?g* (*Neg p*)›
using *calculation* **by** *simp*
moreover have ‹*list-all* (*eval e ?f ?g*) *ps*›
using *calculation* **by** *simp*
then have ‹*eval e ?f ?g p*›
using *mod* **unfolding** *model-def* **by** *blast*
ultimately show *False* **by** *simp*
qed

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* (− (⋃ *p* ∈ *S*. *params p*)) ∧ (∃ *f*. ∀ (*p*::(*'a*, *'b*)*form*) ∈ *S*. *eval e f g p*)}›
unfolding *consistency-def*
proof (*intro allI impI conjI*)
let *?C* = ‹{*S*. *infinite* (− (⋃ *p* ∈ *S*. *params p*)) ∧ (∃ *f*. ∀ *p* ∈ *S*. *eval e f g p*)}›

fix *S* :: ‹(*'a*, *'b*) *form set*›
assume ‹*S* ∈ *?C*›
then have *inf-params*: ‹*infinite* (− (⋃ *p* ∈ *S*. *params p*))›
and ‹∃ *f*. ∀ *p* ∈ *S*. *eval e f g p*›
by *blast*+

then obtain f **where** $*$: ‹$\forall x \in S.\ eval\ e\ f\ g\ x$› **by** *blast*

{ fix $p\ ts$
show ‹$\neg\ (Pred\ p\ ts \in S \wedge Neg\ (Pred\ p\ ts) \in S)$›
proof
assume ‹$Pred\ p\ ts \in S \wedge Neg\ (Pred\ p\ ts) \in S$›
then have ‹$eval\ e\ f\ g\ (Pred\ p\ ts) \wedge eval\ e\ f\ g\ (Neg\ (Pred\ p\ ts))$›
using $*$ **by** *blast*
then show *False* **by** *simp*
qed }

show ‹$FF \notin S$›
using $*$ **by** *fastforce*

show ‹$Neg\ TT \notin S$›
using $*$ **by** *fastforce*

{ fix Z
assume ‹$Neg\ (Neg\ Z) \in S$›
then have ‹$\forall x \in S \cup \{Neg\ (Neg\ Z)\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$\forall x \in S \cup \{Z\}.\ eval\ e\ f\ g\ x$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{Z\}.\ params\ p))$›
using *inf-params* **by** *simp*
ultimately show ‹$S \cup \{Z\} \in ?C$›
by *blast* **}**

{ fix $A\ B$
assume ‹$And\ A\ B \in S$›
then have ‹$\forall x \in S \cup \{And\ A\ B\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$\forall x \in S \cup \{A, B\}.\ eval\ e\ f\ g\ x$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{A, B\}.\ params\ p))$›
using *inf-params* **by** *simp*
ultimately show ‹$S \cup \{A, B\} \in ?C$›
by *blast* **}**

{ fix $A\ B$
assume ‹$Neg\ (Or\ A\ B) \in S$›
then have ‹$\forall x \in S \cup \{Neg\ (Or\ A\ B)\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$\forall x \in S \cup \{Neg\ A, Neg\ B\}.\ eval\ e\ f\ g\ x$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{Neg\ A, Neg\ B\}.\ params\ p))$›
using *inf-params* **by** *simp*
ultimately show ‹$S \cup \{Neg\ A, Neg\ B\} \in ?C$›
by *blast* **}**

{ fix A B
assume ‹$Neg\ (Impl\ A\ B) \in S$›
then have ‹$\forall x \in S \cup \{Neg\ (Impl\ A\ B)\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$\forall x \in S \cup \{A,\ Neg\ B\}.\ eval\ e\ f\ g\ x$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{A,\ Neg\ B\}.\ params\ p))$›
using *inf-params* **by** *simp*
ultimately show ‹$S \cup \{A,\ Neg\ B\} \in ?C$›
by *blast* **}**

{ fix A B
assume ‹$Or\ A\ B \in S$›
then have ‹$\forall x \in S \cup \{Or\ A\ B\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$(\forall x \in S \cup \{A\}.\ eval\ e\ f\ g\ x) \vee$
$(\forall x \in S \cup \{B\}.\ eval\ e\ f\ g\ x)$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{A\}.\ params\ p))$›
and ‹$infinite\ (-\ (\bigcup p \in S \cup \{B\}.\ params\ p))$›
using *inf-params* **by** *simp-all*
ultimately show ‹$S \cup \{A\} \in ?C \vee S \cup \{B\} \in ?C$›
by *blast* **}**

{ fix A B
assume ‹$Neg\ (And\ A\ B) \in S$›
then have ‹$\forall x \in S \cup \{Neg\ (And\ A\ B)\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$(\forall x \in S \cup \{Neg\ A\}.\ eval\ e\ f\ g\ x) \vee$
$(\forall x \in S \cup \{Neg\ B\}.\ eval\ e\ f\ g\ x)$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{Neg\ A\}.\ params\ p))$›
and ‹$infinite\ (-\ (\bigcup p \in S \cup \{Neg\ B\}.\ params\ p))$›
using *inf-params* **by** *simp-all*
ultimately show ‹$S \cup \{Neg\ A\} \in ?C \vee S \cup \{Neg\ B\} \in ?C$›
by *blast* **}**

{ fix A B
assume ‹$Impl\ A\ B \in S$›
then have ‹$\forall x \in S \cup \{Impl\ A\ B\}.\ eval\ e\ f\ g\ x$›
using $*$ **by** *blast*
then have ‹$(\forall x \in S \cup \{Neg\ A\}.\ eval\ e\ f\ g\ x) \vee$
$(\forall x \in S \cup \{B\}.\ eval\ e\ f\ g\ x)$›
by *simp*
moreover have ‹$infinite\ (-\ (\bigcup p \in S \cup \{Neg\ A\}.\ params\ p))$›
and ‹$infinite\ (-\ (\bigcup p \in S \cup \{B\}.\ params\ p))$›
using *inf-params* **by** *simp-all*
ultimately show ‹$S \cup \{Neg\ A\} \in ?C \vee S \cup \{B\} \in ?C$›

by *blast* **}**

{ fix P **and** t :: ‹$'a$ *term*› **assume** ‹*Forall* $P \in S$› **then have** ‹$\forall x \in S \cup \{$*Forall* $P\}$. *eval* e f g x› **using** $*$ **by** *blast* **then have** ‹$\forall x \in S \cup \{P[t/0]\}$. *eval* e f g x› **by** *simp* **moreover have** ‹*infinite* $(- (\bigcup p \in S \cup \{P[t/0]\}$. *params* $p))$› **using** *inf-params* **by** *simp* **ultimately show** ‹$S \cup \{P[t/0]\} \in ?C$› **by** *blast* **}**

{ fix P **and** t :: ‹$'a$ *term*› **assume** ‹*Neg* (*Exists* P) $\in S$› **then have** ‹$\forall x \in S \cup \{$*Neg* (*Exists* P)$\}$. *eval* e f g x› **using** $*$ **by** *blast* **then have** ‹$\forall x \in S \cup \{$*Neg* $(P[t/0])\}$. *eval* e f g x› **by** *simp* **moreover have** ‹*infinite* $(- (\bigcup p \in S \cup \{$*Neg* $(P[t/0])\}$. *params* $p))$› **using** *inf-params* **by** *simp* **ultimately show** ‹$S \cup \{$*Neg* $(P[t/0])\} \in ?C$› **by** *blast* **}**

{ fix P **assume** ‹*Exists* $P \in S$› **then have** ‹$\forall x \in S \cup \{$*Exists* $P\}$. *eval* e f g x› **using** $*$ **by** *blast* **then have** ‹*eval* e f g (*Exists* P)› **by** *blast* **then obtain** z **where** ‹*eval* $(e\langle 0{:}z\rangle)$ f g P› **by** *auto* **moreover obtain** x **where** $**$: ‹$x \in - (\bigcup p \in S$. *params* $p)$› **using** *inf-params infinite-imp-nonempty* **by** *blast* **then have** ‹$x \notin$ *params* P› **using** ‹*Exists* $P \in S$› **by** *auto* **ultimately have** ‹*eval* $(e\langle 0{:}(f(x := \lambda y.\ z))\ x\ []\rangle)$ $(f(x := \lambda y.\ z))$ g P› **by** *simp* **moreover have** ‹$\forall p \in S$. *eval* e $(f(x := \lambda y.\ z))$ g p› **using** $*$ $**$ **by** *simp* **moreover have** ‹*infinite* $(- (\bigcup p \in S \cup \{P[$*App* x $[]/0]\}$. *params* $p))$› **using** *inf-params* **by** *simp* **ultimately have** ‹$S \cup \{P[$*App* x $[]/0]\}$ $\in$ $\{S$. *infinite* $(- (\bigcup p \in S$. *params* $p)) \wedge (\forall p \in S$. *eval* e $(f(x := \lambda y.\ z))$ g $p)\}$› **by** *simp* **then show** ‹$\exists x$. $S \cup \{P[$*App* x $[]/0]\} \in ?C$› **by** *blast* **}**

```
  { fix P
    assume ‹Neg (Forall P) ∈ S›
    then have ‹∀ x ∈ S ∪ {Neg (Forall P)}. eval e f g x›
      using ∗ by blast
    then have ‹eval e f g (Neg (Forall P))›
      by blast
    then obtain z where ‹¬ eval (e⟨0:z⟩) f g P›
      by auto
    moreover obtain x where ∗∗: ‹x ∈ − (⋃ p ∈ S. params p)›
      using inf-params infinite-imp-nonempty by blast
    then have ‹x ∉ params P›
      using ‹Neg (Forall P) ∈ S› by auto
    ultimately have ‹¬ eval (e⟨0:(f(x := λy. z)) x []⟩) (f(x := λy. z)) g P›
      by simp
    moreover have ‹∀ p ∈ S. eval e (f(x := λy. z)) g p›
      using ∗ ∗∗ by simp
    moreover have ‹infinite (− (⋃ p ∈ S ∪ {P[App x []/0]}. params p))›
      using inf-params by simp
    ultimately have ‹S ∪ {Neg (P[App x []/0])} ∈
                    {S. infinite (− (⋃ p ∈ S. params p)) ∧ (∀ p ∈ S. eval e (f(x :=
λy. z)) g p)}›
      by simp
    then show ‹∃ x. S ∪ {Neg (P[App x []/0])} ∈ ?C›
      by blast }
qed

theorem doublep-infinite-params:
  ‹infinite (− (⋃ p ∈ psubst (λn::nat. 2 ∗ n) ‘ S. params p))›
proof (rule infinite-super)
  show ‹infinite (range (λn::nat. 2 ∗ n + 1))›
   using inj-onI Suc-1 Suc-mult-cancel1 add-right-imp-eq finite-imageD infinite-UNIV-char-0
    by (metis (no-types, lifting))
next
  have ‹⋀m n. Suc (2 ∗ m) ≠ 2 ∗ n› by arith
  then show ‹range (λn. 2 ∗ n + 1)
    ⊆ − (⋃ p::(nat, 'a) form ∈ psubst (λn . 2 ∗ n) ‘ S. params p)›
    by auto
qed
```

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

```
theorem loewenheim-skolem:
  assumes evalS: ‹∀ p ∈ S. eval e f g p›
  shows ‹∀ p ∈ S. closed 0 p ⟶ eval e' (λn. HApp (2∗n)) (λa ts.
      Pred a (terms-of-hterms ts) ∈ Extend (psubst (λn. 2 ∗ n) ‘ S)
        (mk-finite-char (mk-alt-consistency (close
            {S. infinite (− (⋃ p ∈ S. params p)) ∧ (∃ f. ∀ p ∈ S. eval e f g p)}))))
```

from-nat) *p*›
(**is** ‹∀ - ∈ -. - - - ⟶ *eval* - - *?g* -›)
using *evalS*
proof (*intro ballI impI*)
fix *p*

let *?C* = ‹{*S*. *infinite* (− (⋃*p* ∈ *S*. *params p*)) ∧ (∃*f*. ∀*x* ∈ *S*. *eval e f g x*)}›

assume ‹*p* ∈ *S*›
and ‹*closed 0 p*›
then have ‹*eval e f g p*›
using *evalS* **by** *blast*
then have ‹∀*x* ∈ *S*. *eval e f g x*›
using *evalS* **by** *blast*
then have ‹∀*p* ∈ *psubst* (λ*n*. *2* ∗ *n*) ‘ *S*. *eval e* (λ*n*. *f* (*n div 2*)) *g p*›
by (*simp add*: *psubst-eval*)
then have ‹*psubst* (λ*n*. *2* ∗ *n*) ‘ *S* ∈ *?C*›
using *doublep-infinite-params* **by** *blast*
moreover have ‹*psubst* (λ*n*. *2* ∗ *n*) *p* ∈ *psubst* (λ*n*. *2* ∗ *n*) ‘ *S*›
using ‹*p* ∈ *S*› **by** *blast*
moreover have ‹*closed 0* (*psubst* (λ*n*. *2* ∗ *n*) *p*)›
using ‹*closed 0 p*› **by** *simp*
moreover have ‹*consistency ?C*›
using *sat-consistency* **by** *blast*
ultimately have ‹*eval e′ HApp ?g* (*psubst* (λ*n*. *2* ∗ *n*) *p*)›
using *model-existence* **by** *blast*
then show ‹*eval e′* (λ*n*. *HApp* (*2* ∗ *n*)) *?g p*›
using *psubst-eval* **by** *blast*
qed

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*)›
by (*induct t* **and** *l rule*: *paramst.induct paramsts.induct*) *auto*

lemma *new-psubst-image*: ‹*new c p* ⟹ *d* ∉ *image f* (*params p*) ⟹ *new d* (*psubst*

```
(f(c := d)) p)›
  using new-psubst-image' by (induct p) auto

lemma news-psubst: ‹news c z ⟹ d ∉ image f (⋃p ∈ set z. params p) ⟹
    news d (map (psubst (f(c := d))) z)›
  using new-psubst-image by (induct z) auto

lemma member-psubst: ‹p ∈ set z ⟹ psubst f p ∈ set (map (psubst f) z)›
  by (induct z) auto

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 (induct z p arbitrary: f rule: deriv.induct)
  case (Assum a G)
  then show ?case
    using deriv.Assum member-psubst by blast
next
  case (TTI G)
  then show ?case
    using deriv.TTI by auto
next
  case (FFE G a)
  then show ?case
    using deriv.FFE by auto
next
  case (NegI a G)
  then show ?case
    using deriv.NegI by auto
next
  case (NegE G a)
  then show ?case
    using deriv.NegE by auto
next
  case (Class a G)
  then show ?case
    using deriv.Class by auto
next
  case (ImplE G a b)
  then have ‹map (psubst f) G ⊢ Impl (psubst f a) (psubst f b)›
    and ‹map (psubst f) G ⊢ psubst f a›
    by simp-all
  then show ?case
    using deriv.ImplE by blast
next
  case (ImplI G a b)
  then show ?case
    using deriv.ImplI by auto
```

next
case (*OrE G a b c*)
then have ‹*map* (*psubst f*) *G* ⊢ *Or* (*psubst f a*) (*psubst f b*)›
and ‹*psubst f a* # *map* (*psubst f*) *G* ⊢ *psubst f c*›
and ‹*psubst f b* # *map* (*psubst f*) *G* ⊢ *psubst f c*›
by *simp-all*
then show *?case*
using *deriv.OrE* **by** *blast*
next
case (*OrI1 G a b*)
then show *?case*
using *deriv.OrI1* **by** *auto*
next
case (*OrI2 G a b*)
then show *?case*
using *deriv.OrI2* **by** *auto*
next
case (*AndE1 G a b*)
then show *?case*
using *deriv.AndE1* **by** *auto*
next
case (*AndE2 p q z*)
then show *?case*
using *deriv.AndE2* **by** *auto*
next
case (*AndI G a b*)
then show *?case*
using *deriv.AndI* **by** *fastforce*
next
case (*ExistsE z p c q*)
let *?params* = ‹*params p* ∪ *params q* ∪ (⋃ *p* ∈ *set z*. *params p*)›

have ‹*finite ?params*›
by *simp*
then obtain *fresh* **where** ∗: ‹*fresh* ∉ *?params* ∪ {*c*} ∪ *image f ?params*›
using *ex-new-if-finite inf-params*
by (*metis finite.emptyI finite.insertI finite-UnI finite-imageI*)

let *?f* = ‹*f*(*c* := *fresh*)›

have ‹*news c* (*p* # *q* # *z*)›
using *ExistsE* **by** *simp*
then have ‹*new fresh* (*psubst ?f p*)› ‹*new fresh* (*psubst ?f q*)› ‹*news fresh* (*map* (*psubst ?f*) *z*)›
using ∗ *new-psubst-image news-psubst* **by** (*fastforce simp add*: *image-Un*)+
then have ‹*map* (*psubst ?f*) *z* = *map* (*psubst f*) *z*›
using *ExistsE* **by** (*metis* (*mono-tags*, *lifting*) *Ball-set map-eq-conv psubst-upd*)

have ‹*map* (*psubst ?f*) *z* ⊢ *psubst ?f* (*Exists p*)›

```
    using ExistsE by blast
  then have ‹map (psubst ?f) z ⊢ Exists (psubst ?f p)›
    by simp
  moreover have ‹map (psubst ?f) (subst p (App c []) 0 # z) ⊢ psubst ?f q›
    using ExistsE by blast
  then have ‹subst (psubst ?f p) (App fresh []) 0 # map (psubst ?f) z ⊢ psubst ?f
q›
    by simp
  moreover have ‹news fresh (map (psubst ?f) (p # q # z))›
     using ‹new fresh (psubst ?f p)› ‹new fresh (psubst ?f q)› ‹news fresh (map
(psubst ?f) z)›
    by simp
  then have ‹new fresh (psubst ?f p)› ‹new fresh (psubst ?f q)› ‹news fresh (map
(psubst ?f) z)›
    by simp-all
  ultimately have ‹map (psubst ?f) z ⊢ psubst ?f q›
    using deriv.ExistsE by metis
  then show ?case
    using ExistsE ‹map (psubst ?f) z = map (psubst f) z› by simp
next
  case (ExistsI z p t)
  then show ?case
    using deriv.ExistsI by auto
next
  case (ForallE z p t)
  then show ?case
    using deriv.ForallE by auto
next
  case (ForallI z p c)
  let ?params = ‹params p ∪(⋃p ∈ set z. params p)›

  have ‹finite ?params›
    by simp
  then obtain fresh where ∗: ‹fresh ∉ ?params ∪ {c} ∪ image f ?params›
    using ex-new-if-finite inf-params
    by (metis finite.emptyI finite.insertI finite-UnI finite-imageI)

  let ?f = ‹f(c := fresh)›

  have ‹news c (p # z)›
    using ForallI by simp
  then have ‹new fresh (psubst ?f p)› ‹news fresh (map (psubst ?f) z)›
    using ∗ new-psubst-image news-psubst by (fastforce simp add: image-Un)+
  then have ‹map (psubst ?f) z = map (psubst f) z›
    using ForallI by (metis (mono-tags, lifting) Ball-set map-eq-conv psubst-upd)

  have ‹map (psubst ?f) z ⊢ psubst ?f (subst p (App c []) 0)›
    using ForallI by blast
  then have ‹map (psubst ?f) z ⊢ subst (psubst ?f p) (App fresh []) 0›
```

by *simp*
moreover have ‹*news fresh* (*map* (*psubst ?f*) (*p* # *z*))›
using ‹*new fresh* (*psubst ?f p*)› ‹*news fresh* (*map* (*psubst ?f*) *z*)›
by *simp*
then have ‹*new fresh* (*psubst ?f p*)› ‹*news fresh* (*map* (*psubst ?f*) *z*)›
by *simp-all*
ultimately have ‹*map* (*psubst ?f*) *z* ⊢ *Forall* (*psubst ?f p*)›
using *deriv.ForallI* **by** *metis*
then show *?case*
using *ForallI* ‹*map* (*psubst ?f*) *z* = *map* (*psubst f*) *z*› **by** *simp*
qed

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*›
by (*induct t* **and** *l rule*: *substt.induct substs.induct*) *simp-all*

lemma *params-lift* [*simp*]:
fixes *t* :: ‹'*a term*› **and** *ts* :: ‹'*a term list*›
shows
‹*paramst* (*liftt t*) = *paramst t*›
‹*paramsts* (*liftts ts*) = *paramsts ts*›
by (*induct t* **and** *ts rule*: *paramst.induct paramsts.induct*) *simp-all*

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*)›
by (*induct t* **and** *l rule*: *substt.induct substs.induct*) *simp-all*

lemma *subst-new* [*simp*]: ‹*new-term c s* ⟹ *new c p* ⟹ *new c* (*subst p s m*)›
by (*induct p arbitrary*: *m s*) *simp-all*

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*›
using *assms* **by** (*induct cs*) *auto*

lemma *subc-new′* [*simp*]:
‹*new-term c t* ⟹ *subc-term c s t* = *t*›
‹*new-list c l* ⟹ *subc-list c s l* = *l*›
by (*induct t* **and** *l rule*: *subc-term.induct subc-list.induct*) *auto*

lemma *subc-new* [*simp*]: ‹*new c p* ⟹ *subc c s p* = *p*›
by (*induct p arbitrary*: *s*) *simp-all*

lemma *subcs-news*: ‹*news c z* ⟹ *subcs c s z* = *z*›
by (*induct z*) *simp-all*

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*) ⟹
psubsts f (*subc-list c s l*) = *subc-list* (*f c*) (*psubstt f s*) (*psubsts f l*)›
by (*induct t* **and** *l rule*: *psubstt.induct psubsts.induct*) *simp-all*

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*)›
by (*induct p arbitrary*: *s*) *simp-all*

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*)›
by (*induct z*) (*simp-all add*: *subc-psubst*)

lemma *new-lift*:
‹*new-term c t* ⟹ *new-term c* (*liftt t*)›
‹*new-list c l* ⟹ *new-list c* (*lifts l*)›
by (*induct t* **and** *l rule*: *liftt.induct lifts.induct*) *simp-all*

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*)›
by (*induct t* **and** *l rule*: *substt.induct substs.induct*) *simp-all*

lemma *new-subc* [*simp*]: ‹*new-term d s* ⟹ *new d p* ⟹ *new d* (*subc c s p*)›
by (*induct p arbitrary*: *s*) *simp-all*

lemma *news-subcs*: ‹*new-term d s* ⟹ *news d z* ⟹ *news d* (*subcs c s z*)›
by (*induct z*) *simp-all*

lemma *psubst-new-free'*:
‹*c* ≠ *n* ⟹ *new-term n* (*psubstt* (*id*(*n* := *c*)) *t*)›
‹*c* ≠ *n* ⟹ *new-list n* (*psubstts* (*id*(*n* := *c*)) *l*)›
by (*induct t* **and** *l rule*: *paramst.induct paramsts.induct*) *simp-all*

lemma *psubst-new-free*: ‹*c* ≠ *n* ⟹ *new n* (*psubst* (*id*(*n* := *c*)) *p*)›
using *psubst-new-free'* **by** (*induct p*) *fastforce+*

lemma *map-psubst-new-free*: ‹*c* ≠ *n* ⟹ *news n* (*map* (*psubst* (*id*(*n* := *c*))) *z*)›
using *psubst-new-free* **by** (*induct z*) *fastforce+*

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*›
by (*induct t* **and** *l rule*: *psubstt.induct psubstts.induct*) *auto*

lemma *psubst-new-away* [*simp*]: ‹*new fresh p* ⟹ *psubst* (*id*(*fresh* := *c*)) (*psubst* (*id*(*c* := *fresh*)) *p*) = *p*›
by (*induct p*) *simp-all*

lemma *map-psubst-new-away*:
‹*news fresh z* ⟹ *map* (*psubst* (*id*(*fresh* := *c*))) (*map* (*psubst* (*id*(*c* := *fresh*))) *z*) = *z*›
by (*induct z*) *simp-all*

lemma *psubst-new'*:
‹*new-term c t* ⟹ *psubstt* (*id*(*c* := *x*)) *t* = *t*›
‹*new-list c l* ⟹ *psubstts* (*id*(*c* := *x*)) *l* = *l*›
by (*induct t* **and** *l rule*: *psubstt.induct psubstts.induct*) *auto*

lemma *psubst-new*: ‹*new c p* ⟹ *psubst* (*id*(*c* := *x*)) *p* = *p*›
using *psubst-new'* **by** (*induct p*) *fastforce+*

lemma *map-psubst-new*: ‹*news c z* ⟹ *map* (*psubst* (*id*(*c* := *x*))) *z* = *z*›
using *psubst-new* **by** (*induct z*) *auto*

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*)›
by (*induct t* **and** *l rule*: *substt.induct substs.induct*) *simp-all*

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*)›
by (*induct t* **and** *l rule*: *subc-term.induct subc-list.induct*) *simp-all*

lemma *new-subc-same*: ‹*new-term c s* ⟹ *new c* (*subc c s p*)›
by (*induct p arbitrary*: *s*) *simp-all*

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*)›
by (*induct t* **and** *l rule*: *liftt.induct liftts.induct*) *simp-all*

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*)›
by (*induct t* **and** *l rule*: *subc-term.induct subc-list.induct*) *simp-all*

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 (*induct p arbitrary*: *s m t*)
case *FF*
then show *?case*
by *simp*
next
case *TT*
then show *?case*
by *simp*
next
case (*Pred i l*)
then show *?case*
using *new-subc-put'* **by** *fastforce*
next
case (*Neg p*)
then show *?case*
by (*metis subc.simps*(*4*) *subst.simps*(*7*))
next
case (*Impl p q*)
then show *?case*
by (*metis subc.simps*(*5*) *subst.simps*(*6*))
next
case (*Or p q*)
then show *?case*
by (*metis subc.simps*(*6*) *subst.simps*(*5*))
next
case (*And p q*)
then show *?case*
by (*metis subc.simps*(*7*) *subst.simps*(*4*))

next
case (*Exists p*)
have ‹*subc c s* (*subst* (*Exists p*) (*subc-term c s t*) *m*) =
Exists (*subc c* (*liftt s*) (*subst p* (*subc-term c* (*liftt s*) (*liftt t*)) (*Suc m*)))›
by (*simp add*: *lift-subc*)
also have ‹. . . = *Exists* (*subc c* (*liftt s*) (*subst p* (*liftt t*) (*Suc m*)))›
using *Exists new-lift*(*1*) **by** *metis*
finally show *?case*
by *simp*
next
case (*Forall p*)
have ‹*subc c s* (*subst* (*Forall p*) (*subc-term c s t*) *m*) =
Forall (*subc c* (*liftt s*) (*subst p* (*subc-term c* (*liftt s*) (*liftt t*)) (*Suc m*)))›
by (*simp add*: *lift-subc*)
also have ‹. . . = *Forall* (*subc c* (*liftt s*) (*subst p* (*liftt t*) (*Suc m*)))›
using *Forall new-lift*(*1*) **by** *metis*
finally show *?case*
by *simp*
qed

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*›
by (*induct t* **and** *l rule*: *subc-term.induct subc-list.induct*) *simp-all*

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*›
using *subc-subst-new′* **by** (*induct p arbitrary*: *m t s*) *fastforce+*

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*›
using *subc-subst-new subst-0-lift*(*1*) **by** *metis*

lemma *member-subc*: ‹*p* ∈ *set z* ⟹ *subc c s p* ∈ *set* (*subcs c s z*)›
by (*induct z*) *auto*

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 (*induct z p arbitrary*: *c s rule*: *deriv.induct*)
case (*Assum p z*)
then show *?case*
using *member-subc deriv.Assum* **by** *fast*
next
case *TTI*
then show *?case*

```
    using deriv.TTI by simp
  case FFE
  then show ?case
    using deriv.FFE by auto
next
  case (NegI z p)
  then show ?case
    using deriv.NegI by auto
next
  case (NegE z p)
  then show ?case
    using deriv.NegE by fastforce
next
  case (Class p z)
  then show ?case
    using deriv.Class by auto
next
  case (ImplE z p q)
  then show ?case
    using deriv.ImplE by fastforce
next
  case (ImplI z q p)
  then show ?case
    using deriv.ImplI by fastforce
next
  case (OrE z p q r)
  then show ?case
    using deriv.OrE by fastforce
next
  case (OrI1 z p q)
  then show ?case
    using deriv.OrI1 by fastforce
next
  case (OrI2 z q p)
  then show ?case
    using deriv.OrI2 by fastforce
next
  case (AndE1 z p q)
  then show ?case
    using deriv.AndE1 by fastforce
next
  case (AndE2 z p q)
  then show ?case
    using deriv.AndE2 by fastforce
next
  case (AndI p z q)
  then show ?case
    using deriv.AndI by fastforce
next
```

case (*ExistsE z p d q*)
then show *?case*
proof (*cases* ‹*c* = *d*›)
case *True*
then have ‹*z* ⊢ *q*›
using *ExistsE deriv.ExistsE* **by** *fast*
moreover have ‹*new c q*› **and** ‹*news c z*›
using *ExistsE True* **by** *simp-all*
ultimately show *?thesis*
using *subc-new subcs-news* **by** *metis*
next
case *False*
let *?params* = ‹*params p* ∪ *params q* ∪ (⋃ *p* ∈ *set z*. *params p*) ∪ *paramst s* ∪ {*c*} ∪ {*d*}›

have ‹*finite ?params*›
by *simp*
then obtain *fresh* **where** *fresh*: ‹*fresh* ∉ *?params*›
using *inf-params* **by** (*meson ex-new-if-finite infinite-UNIV-listI*)

let *?s* = ‹*psubstt* (*id*(*d* := *fresh*)) *s*›
let *?f* = ‹*id*(*d* := *fresh*, *fresh* := *d*)›

have *f*: ‹∀ *x* ∈ *?params*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›
using *fresh* **by** *simp*

have ‹*new-term d ?s*›
using *fresh psubst-new-free*′(*1*) **by** *fast*
then have ‹*psubstt ?f ?s* = *psubstt* (*id*(*fresh* := *d*)) *?s*›
by (*metis fun-upd-twist psubstt-upd*(*1*))
then have *psubst-s*: ‹*psubstt ?f ?s* = *s*›
using *fresh* **by** *simp*

have ‹*?f c* = *c*› **and** ‹*new-term* (*?f c*) (*App fresh* []›
using *False fresh* **by** *auto*

have ‹*subcs c* (*psubstt ?f ?s*) *z* ⊢ *subc c* (*psubstt ?f ?s*) (*Exists p*)›
using *ExistsE* **by** *blast*
then have *exi-p*:
‹*subcs c s z* ⊢ *Exists* (*subc c* (*liftt* (*psubstt ?f ?s*)) *p*)›
using *psubst-s* **by** *simp*

have ‹*news d z*›
using *ExistsE* **by** *simp*
moreover have ‹*news fresh z*›
using *fresh* **by** (*induct z*) *simp-all*
ultimately have ‹*map* (*psubst ?f*) *z* = *z*›
by (*induct z*) *simp-all*
moreover have ‹∀ *x* ∈ ⋃ *p* ∈ *set z*. *params p*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›

```
    by auto
  ultimately have psubst-z: ‹map (psubst ?f) (subcs c ?s z) = subcs c s z›
    using ‹?f c = c› psubst-s by (simp add: subcs-psubst)

  have ‹psubst ?f (subc c ?s (subst p (App d []) 0)) =
      subc (?f c) (psubstt ?f ?s) (psubst ?f (subst p (App d []) 0))›
    using fresh by (simp add: subc-psubst)
  also have ‹... = subc c s (subst (psubst ?f p) (App fresh []) 0)›
    using psubst-subst psubst-s ‹?f c = c› by simp
  also have ‹... = subc c s (subst p (App fresh []) 0)›
    using ExistsE fresh by simp
  finally have psubst-p: ‹psubst ?f (subc c ?s (subst p (App d []) 0)) =
      subst (subc c (liftt s) p) (App fresh []) 0›
    using subc-sub-0-new ‹new-term (?f c) (App fresh [])› ‹?f c = c› by metis

  have ‹∀ x ∈ params q. x ≠ c ⟶ ?f x ≠ ?f c›
    using f by blast
  then have psubst-q: ‹psubst ?f (subc c ?s q) = subc c s q›
    using ExistsE fresh ‹?f c = c› psubst-s f by (simp add: subc-psubst)

  have ‹subcs c ?s (subst p (App d []) 0 # z) ⊢ subc c ?s q›
    using ExistsE by blast
  then have ‹subc c ?s (subst p (App d []) 0) # subcs c ?s z ⊢ subc c ?s q›
    by simp
  then have ‹psubst ?f (subc c ?s (subst p (App d []) 0)) # map (psubst ?f)
(subcs c ?s z)
      ⊢ psubst ?f (subc c ?s q)›
    using deriv-psubst inf-params by fastforce
  then have q: ‹subst (subc c (liftt s) p) (App fresh []) 0 # subcs c s z ⊢ subc c
s q›
    using psubst-q psubst-z psubst-p by simp

  have ‹new fresh (subc c (liftt s) p)›
    using fresh new-subc new-lift by simp
  moreover have ‹new fresh (subc c s q)›
    using fresh new-subc by simp
  moreover have ‹news fresh (subcs c s z)›
    using fresh ‹news fresh z› by (simp add: news-subcs)
  ultimately show ‹subcs c s z ⊢ subc c s q›
    using deriv.ExistsE exi-p q psubst-s by metis
 qed
next
 case (ExistsI z p t)
 let ?params = ‹params p ∪ (⋃p ∈ set z. params p) ∪ paramst s ∪ paramst t ∪
{c}›

 have ‹finite ?params›
   by simp
 then obtain fresh where fresh: ‹fresh ∉ ?params›
```

```
    using inf-params by (meson ex-new-if-finite infinite-UNIV-listI)

  let ?f = ‹id(c := fresh)›
  let ?g = ‹id(fresh := c)›
  let ?s = ‹psubstt ?f s›

  have c: ‹?g c = c›
    using fresh by simp
  have s: ‹psubstt ?g ?s = s›
    using fresh by simp
  have p: ‹psubst ?g (Exists p) = Exists p›
    using fresh by simp

  have ‹∀ x ∈ (⋃ p ∈ set z. params p). x ≠ c ⟶ ?g x ≠ ?g c›
    using fresh by auto
  moreover have ‹map (psubst ?g) z = z›
    using fresh by (induct z) simp-all
  ultimately have z: ‹map (psubst ?g) (subcs c ?s z) = subcs c s z›
    using s by (simp add: subcs-psubst)

  have ‹new-term c ?s›
    using fresh psubst-new-free' by fast
  then have ‹subcs c ?s z ⊢ subc c ?s (subst p (subc-term c ?s t) 0)›
    using ExistsI new-subc-put by metis
  moreover have ‹new-term c (subc-term c ?s t)›
    using ‹new-term c ?s› new-subc-same' by fast
  ultimately have ‹subcs c ?s z ⊢ subst (subc c (liftt ?s) p) (subc-term c ?s t) 0›
    using subc-sub-0-new by metis

  then have ‹subcs c ?s z ⊢ subc c ?s (Exists p)›
    using deriv.ExistsI by simp
  then have ‹map (psubst ?g) (subcs c ?s z) ⊢ psubst ?g (subc c ?s (Exists p))›
    using deriv-psubst inf-params by blast
  moreover have ‹∀ x ∈ params (Exists p). x ≠ c ⟶ ?g x ≠ ?g c›
    using fresh by auto
  ultimately show ‹subcs c s z ⊢ subc c s (Exists p)›
    using c s p z by (simp add: subc-psubst)
next
  case (ForallE z p t)
  let ?params = ‹params p ∪ (⋃ p ∈ set z. params p) ∪ paramst s ∪ paramst t ∪
{c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    using inf-params by (meson ex-new-if-finite infinite-UNIV-listI)

  let ?f = ‹id(c := fresh)›
  let ?g = ‹id(fresh := c)›
```

let *?s* = ‹*psubstt ?f s*›

have *c*: ‹*?g c* = *c*›
using *fresh* **by** *simp*
have *s*: ‹*psubstt ?g ?s* = *s*›
using *fresh* **by** *simp*
have *p*: ‹*psubst ?g* (*subst p t 0*) = *subst p t 0*›
using *fresh psubst-new psubst-subst subst-new psubst-new'*(*1*) **by** *fastforce*

have ‹∀ *x* ∈ (⋃ *p* ∈ *set z*. *params p*). *x* ≠ *c* ⟶ *?g x* ≠ *?g c*›
using *fresh* **by** *auto*
moreover have ‹*map* (*psubst ?g*) *z* = *z*›
using *fresh* **by** (*induct z*) *simp-all*
ultimately have *z*: ‹*map* (*psubst ?g*) (*subcs c ?s z*) = *subcs c s z*›
using *s* **by** (*simp add*: *subcs-psubst*)

have ‹*new-term c ?s*›
using *fresh psubst-new-free'* **by** *fastforce*

have ‹*subcs c ?s z* ⊢ *Forall* (*subc c* (*liftt ?s*) *p*)›
using *ForallE* **by** *simp*
then have ‹*subcs c ?s z* ⊢ *subst* (*subc c* (*liftt ?s*) *p*) (*subc-term c ?s t*) *0*›
using *deriv.ForallE* **by** *blast*
moreover have ‹*new-term c* (*subc-term c ?s t*)›
using ‹*new-term c ?s*› *new-subc-same'* **by** *fast*
ultimately have ‹*subcs c ?s z* ⊢ *subc c ?s* (*subst p* (*subc-term c ?s t*) *0*)›
by *simp*
then have ‹*subcs c ?s z* ⊢ *subc c ?s* (*subst p t 0*)›
using *new-subc-put* ‹*new-term c ?s*› **by** *metis*
then have ‹*map* (*psubst ?g*) (*subcs c ?s z*) ⊢ *psubst ?g* (*subc c ?s* (*subst p t 0*))›
using *deriv-psubst inf-params* **by** *blast*
moreover have ‹∀ *x* ∈ *params* (*subst p t 0*). *x* ≠ *c* ⟶ *?g x* ≠ *?g c*›
using *fresh p psubst-new-free* **by** (*metis fun-upd-apply id-apply*)
ultimately show ‹*subcs c s z* ⊢ *subc c s* (*subst p t 0*)›
using *c s p z* **by** (*simp add*: *subc-psubst*)
next
case (*ForallI z p d*)
then show *?case*
proof (*cases* ‹*c* = *d*›)
case *True*
then have ‹*z* ⊢ *Forall p*›
using *ForallI deriv.ForallI* **by** *fast*
moreover have ‹*new c p*› **and** ‹*news c z*›
using *ForallI True* **by** *simp-all*
ultimately show *?thesis*
by (*simp add*: *subcs-news*)
next
case *False*
let *?params* = ‹*params p* ∪ (⋃ *p* ∈ *set z*. *params p*) ∪ *paramst s* ∪ {*c*} ∪ {*d*}›

have ‹*finite ?params*›
by *simp*
then obtain *fresh* **where** *fresh*: ‹*fresh* ∉ *?params*›
using *inf-params* **by** (*meson ex-new-if-finite infinite-UNIV-listI*)

let *?s* = ‹*psubstt* (*id*(*d* := *fresh*)) *s*›
let *?f* = ‹*id*(*d* := *fresh*, *fresh* := *d*)›

have *f*: ‹∀ *x* ∈ *?params*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›
using *fresh* **by** *simp*

have ‹*new-term d ?s*›
using *fresh psubst-new-free'* **by** *fastforce*
then have ‹*psubstt ?f ?s* = *psubstt* (*id*(*fresh* := *d*)) *?s*›
by (*metis fun-upd-twist psubstt-upd*(*1*))
then have *psubst-s*: ‹*psubstt ?f ?s* = *s*›
using *fresh* **by** *simp*

have ‹*?f c* = *c*› **and** ‹*new-term c* (*App fresh* [])›
using *False fresh* **by** *auto*

have ‹*psubst ?f* (*subc c ?s* (*subst p* (*App d* []) *0*)) =
subc (*?f c*) (*psubstt ?f ?s*) (*psubst ?f* (*subst p* (*App d* []) *0*))›
by (*simp add*: *subc-psubst*)
also have ‹. . . = *subc c s* (*subst* (*psubst ?f p*) (*App fresh* []) *0*)›
using ‹*?f c* = *c*› *psubst-subst psubst-s* **by** *simp*
also have ‹. . . = *subc c s* (*subst p* (*App fresh* []) *0*)›
using *ForallI fresh* **by** *simp*
finally have *psubst-p*: ‹*psubst ?f* (*subc c ?s* (*subst p* (*App d* []) *0*)) =
subst (*subc c* (*liftt s*) *p*) (*App fresh* []) *0*›
using *subc-sub-0-new* ‹*new-term c* (*App fresh* []›) **by** *simp*

have ‹*news d z*›
using *ForallI* **by** *simp*
moreover have ‹*news fresh z*›
using *fresh* **by** (*induct z*) *simp-all*
ultimately have ‹*map* (*psubst ?f*) *z* = *z*›
by (*induct z*) *simp-all*
moreover have ‹∀ *x* ∈ ⋃ *p* ∈ *set z*. *params p*. *x* ≠ *c* ⟶ *?f x* ≠ *?f c*›
by *auto*
ultimately have *psubst-z*: ‹*map* (*psubst ?f*) (*subcs c ?s z*) = *subcs c s z*›
using ‹*?f c* = *c*› *psubst-s* **by** (*simp add*: *subcs-psubst*)

have ‹*subcs c ?s z* ⊢ *subc c ?s* (*subst p* (*App d* []) *0*)›
using *ForallI* **by** *blast*
then have ‹*map* (*psubst ?f*) (*subcs c ?s z*) ⊢ *psubst ?f* (*subc c ?s* (*subst p* (*App d* []) *0*))›
using *deriv-psubst inf-params* **by** *blast*

```
    then have ‹subcs c s z ⊢ psubst ?f (subc c ?s (subst p (App d []) 0))›
      using psubst-z by simp
    then have sub-p: ‹subcs c s z ⊢ subst (subc c (liftt s) p) (App fresh []) 0›
      using psubst-p by simp

    have ‹new-term fresh s›
      using fresh by simp
    then have ‹new-term fresh (liftt s)›
      using new-lift by simp
    then have ‹new fresh (subc c (liftt s) p)›
      using fresh new-subc by simp
    moreover have ‹news fresh (subcs c s z)›
      using ‹news fresh z› ‹new-term fresh s› news-subcs by fast
    ultimately show ‹subcs c s z ⊢ subc c s (Forall p)›
      using deriv.ForallI sub-p by simp
  qed
qed
```

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')›
  using assms by force

lemma subset-cons: ‹set z ⊆ set z' ⟹ set (p # z) ⊆ set (p # z')›
  by auto

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 (induct z p arbitrary: z' rule: deriv.induct)
  case (Assum p z)
  then show ?case
    using deriv.Assum by auto
next
  case TTI
  then show ?case
    using deriv.TTI by auto
next
  case FFE
  then show ?case
    using deriv.FFE by auto
next
  case (NegI p z)
  then show ?case
    using deriv.NegI subset-cons by metis
next
```

```
  case (NegE p z)
  then show ?case
    using deriv.NegE by metis
next
  case (Class z p)
  then show ?case
    using deriv.Class subset-cons by metis
next
  case (ImplE z p q)
  then show ?case
    using deriv.ImplE by blast
next
  case (ImplI z q p)
  then show ?case
    using deriv.ImplI subset-cons by metis
next
  case (OrE z p q z )
  then show ?case
    using deriv.OrE subset-cons by metis
next
  case (OrI1 z p q)
  then show ?case
    using deriv.OrI1 by blast
next
  case (OrI2 z q p)
  then show ?case
    using deriv.OrI2 by blast
next
  case (AndE1 z p q)
  then show ?case
    using deriv.AndE1 by blast
next
  case (AndE2 z p q)
  then show ?case
    using deriv.AndE2 by blast
next
  case (AndI z p q)
  then show ?case
    using deriv.AndI by blast
next
  case (ExistsE z p c q)
  let ?params = ‹params p ∪ params q ∪ (⋃p ∈ set z'. params p) ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    using inf-params by (meson ex-new-if-finite List.finite-set infinite-UNIV-listI)

  let ?z' = ‹map (psubst (id(c := fresh))) z'›
```

```
  have ‹news c z›
    using ExistsE by simp
  then have ‹set z ⊆ set ?z'›
    using ExistsE psubst-new-subset by (simp add: Ball-set)
  then have ‹?z' ⊢ Exists p›
    using ExistsE by blast

  moreover have ‹set (subst p (App c []) 0 # z) ⊆ set (subst p (App c []) 0 #
?z')›
    using ‹set z ⊆ set ?z'› by auto
  then have ‹subst p (App c []) 0 # ?z' ⊢ q›
    using ExistsE by blast

  moreover have ‹news c ?z'›
    using fresh by (simp add: map-psubst-new-free)
  then have ‹new c p› ‹new c q› ‹news c ?z'›
    using ExistsE by simp-all

  ultimately have ‹?z' ⊢ q›
    using ExistsE deriv.ExistsE by metis

  then have ‹map (psubst (id(fresh := c))) ?z' ⊢ psubst (id(fresh := c)) q›
    using deriv-psubst inf-params by blast
  moreover have ‹map (psubst (id(fresh := c))) ?z' = z'›
    using fresh map-psubst-new-away Ball-set by fastforce
  moreover have ‹psubst (id(fresh := c)) q = q›
    using fresh by simp
  ultimately show ‹z' ⊢ q›
    by simp
next
  case (ExistsI z p t)
  then show ?case
    using deriv.ExistsI by blast
next
  case (ForallE p z t)
  then show ?case
    using deriv.ForallE by blast
next
  case (ForallI z p c)
  let ?params = ‹params p ∪ (⋃p ∈ set z'. params p) ∪ {c}›

  have ‹finite ?params›
    by simp
  then obtain fresh where fresh: ‹fresh ∉ ?params›
    using inf-params by (meson ex-new-if-finite List.finite-set infinite-UNIV-listI)

  let ?z' = ‹map (psubst (id(c := fresh))) z'›
```

```
  have ‹news c z›
    using ForallI by simp
  then have ‹set z ⊆ set ?z'›
    using ForallI psubst-new-subset by (metis (no-types, lifting) Ball-set UN-iff)
  then have ‹?z' ⊢ subst p (App c []) 0›
    using ForallI by blast

  moreover have ‹∀ p ∈ set ?z'. c ∉ params p›
    using fresh psubst-new-free by fastforce
  then have ‹list-all (λp. c ∉ params p) (p # ?z')›
    using ForallI by (simp add: list-all-iff)
  then have ‹new c p› ‹news c ?z'›
    by simp-all

  ultimately have ‹?z' ⊢ Forall p›
    using ForallI deriv.ForallI by fast

  then have ‹map (psubst (id(fresh := c))) ?z' ⊢ psubst (id(fresh := c)) (Forall
p)›
    using deriv-psubst inf-params by blast
  moreover have ‹map (psubst (id(fresh := c))) ?z' = z'›
    using fresh map-psubst-new-away Ball-set by fastforce
  moreover have ‹psubst (id(fresh := c)) (Forall p) = Forall p›
    using fresh ForallI by simp
  ultimately show ‹z' ⊢ Forall p›
    by simp
qed
```

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)›
  unfolding model-def by (induct z) auto

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 −
  have ‹set z ⊆ set (p # z)›
    by auto
  then have ‹p # z ⊢ Impl p q›
    using assms weaken-assumptions inf-params by blast
  moreover have ‹p # z ⊢ p›
```

by (*simp add*: *Assum*)
ultimately show ‹*p* # *z* ⊢ *q*›
using *ImplE* **by** *blast*
qed

lemma *remove-imps*:
assumes ‹*infinite* (− *params p*)›
shows ‹*z'* ⊢ *put-imps p z* ⟹ *rev z* @ *z'* ⊢ *p*›
using *assms shift-imp-assum* **by** (*induct z arbitrary*: *z'*) *auto*

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*›
by (*induct t* **and** *l rule*: *substt.induct substs.induct*) *auto*

lemma *subc-sub-closed-var* [*simp*]: ‹*new c p* ⟹ *closed* (*Suc m*) *p* ⟹
subc c (*Var m*) (*subst p* (*App c* []) *m*) = *p*›
by (*induct p arbitrary*: *m*) *simp-all*

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*))›
by (*induct k arbitrary*: *i*) *simp-all*

lemma *closed-put-unis* [*simp*]: ‹*closed m* (*put-unis k p*) = *closed* (*m* + *k*) *p*›
by (*induct k arbitrary*: *m*) *simp-all*

lemma *valid-put-unis*: ‹∀ (*e* :: *nat* ⇒ '*a*) *f g*. *eval e f g p* ⟹
eval (*e* :: *nat* ⇒ '*a*) *f g* (*put-unis m p*)›
by (*induct m arbitrary*: *e*) *simp-all*

lemma *put-unis-collapse*: ‹*put-unis m* (*put-unis n p*) = *put-unis* (*m* + *n*) *p*›
by (*induct m*) *simp-all*

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 (*induct cs arbitrary*: *p*)
case (*Cons c cs*)

```
  then have ‹[] ⊢ Forall (put-unis (length cs) p)›
    by simp
  then have ‹[] ⊢ subst (put-unis (length cs) p) (App c []) 0›
    using ForallE by blast
  then show ?case
    using Cons by simp
qed simp

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›
  using assms deriv-subc by (induct xs arbitrary: p) fastforce+

lemma vars-for-consts-for-unis:
  ‹closed (length cs) p ⟹ list-all (λc. new c p) cs ⟹ distinct cs ⟹
   vars-for-consts (consts-for-unis (put-unis (length cs) p) cs) cs = p›
  by (induct cs arbitrary: p) (simp-all add: subst-new-all)

lemma fresh-constant:
  fixes p :: ‹('a, 'b) form›
  assumes ‹infinite (UNIV :: 'a set)›
  shows ‹∃ c. c ∉ set cs ∧ new c p›
proof −
  have ‹finite (set cs ∪ params p)›
    by simp
  then show ?thesis
    using assms ex-new-if-finite UnI1 UnI2 by metis
qed

lemma fresh-constants:
  fixes p :: ‹('a, 'b) form›
  assumes ‹infinite (UNIV :: 'a set)›
  shows ‹∃ cs. length cs = m ∧ list-all (λc. new c p) cs ∧ distinct cs›
proof (induct m)
  case (Suc m)
  then obtain cs where ‹length cs = m ∧ list-all (λc. new c p) cs ∧ distinct cs›
    by blast
  moreover obtain c where ‹c ∉ set cs ∧ new c p›
    using Suc assms fresh-constant by blast
  ultimately have ‹length (c # cs) = Suc m ∧ list-all (λc. new c p) (c # cs) ∧
distinct (c # cs)›
    by simp
  then show ?case
    by blast
qed simp
```

lemma *closed-max*:
assumes ‹*closed m p*› ‹*closed n q*›
shows ‹*closed* (*max m n*) *p* ∧ *closed* (*max m n*) *q*›
proof −
have ‹*m* ≤ *max m n*› **and** ‹*n* ≤ *max m n*›
by *simp-all*
then show *?thesis*
using *assms closed-mono* **by** *metis*
qed

lemma *ex-closed'* [*simp*]:
fixes *t* :: ‹'*a term*› **and** *l* :: ‹'*a term list*›
shows ‹∃ *m*. *closedt m t*› ‹∃ *n*. *closedts n l*›
proof (*induct t* **and** *l rule*: *closedt.induct closedts.induct*)
case (*Cons-term t l*)
then obtain *m* **and** *n* **where** ‹*closedt m t*› **and** ‹*closedts n l*›
by *blast*
moreover have ‹*m* ≤ *max m n*› **and** ‹*n* ≤ *max m n*›
by *simp-all*
ultimately have ‹*closedt* (*max m n*) *t*› **and** ‹*closedts* (*max m n*) *l*›
using *closedt-mono* **by** *blast*+
then show *?case*
by *auto*
qed *auto*

lemma *ex-closed* [*simp*]: ‹∃ *m*. *closed m p*›
proof (*induct p*)
case *FF*
then show *?case*
by *simp*
next
case *TT*
then show *?case*
by *simp*
next
case (*Neg p*)
then show *?case*
by *simp*
next
case (*Impl p q*)
then show *?case*
using *closed-max* **by** *fastforce*
next
case (*Or p q*)
then show *?case*
using *closed-max* **by** *fastforce*
next
case (*And p q*)

```
  then show ?case
    using closed-max by fastforce
next
  case (Exists p)
  then obtain m where ‹closed m p›
    by blast
  then have ‹closed (Suc m) p›
    using closed-mono Suc-n-not-le-n nat-le-linear by blast
  then show ?case
    by auto
next
  case (Forall p)
  then obtain m where ‹closed m p›
    by blast
  then have ‹closed (Suc m) p›
    using closed-mono Suc-n-not-le-n nat-le-linear by blast
  then show ?case
    by auto
qed simp-all

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

lemma remove-unis-sentence:
  assumes inf-params: ‹infinite (− params p)›
    and ‹closed 0 (put-unis m p)› ‹[] ⊢ put-unis m p›
  shows ‹[] ⊢ p›
proof −
  obtain cs :: ‹'a list› where ‹length cs = m›
    and ∗: ‹distinct cs› and ∗∗: ‹list-all (λc. new c p) cs›
    using assms finite-compl finite-params fresh-constants inf-params by metis
  then have ‹[] ⊢ consts-for-unis (put-unis (length cs) p) cs›
    using assms consts-for-unis by blast
  then have ‹[] ⊢ vars-for-consts (consts-for-unis (put-unis (length cs) p) cs) cs›
    using vars-for-consts inf-params by fastforce
  moreover have ‹closed (length cs) p›
    using assms ‹length cs = m› by simp
  ultimately show ‹[] ⊢ p›
    using vars-for-consts-for-unis ∗ ∗∗ by metis
qed
```

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 −
  let ?p = ‹put-imps p (rev z)›
```

have ∗: ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g ?p*›
using *assms semantics-put-imps* **unfolding** *model-def* **by** *fastforce*
obtain *m* **where** ∗∗: ‹*closed 0* (*put-unis m ?p*)›
using *ex-closure* **by** *blast*
moreover have ‹*list-all* (*closed 0*) []›
by *simp*
moreover have ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *e*, *f*, *g*, [] ⊨ *put-unis m ?p*›
using ∗ *valid-put-unis* **unfolding** *model-def* **by** *blast*
ultimately have ‹[] ⊢ *put-unis m ?p*›
using *natded-complete* **by** *blast*
then have ‹[] ⊢ *?p*›
using ∗∗ *remove-unis-sentence* **by** *fastforce*
then show ‹*z* ⊢ *p*›
using *remove-imps* **by** *fastforce*
qed

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*›
using *completeness correctness*
unfolding *model-def* **by** (*metis list.pred-inject*(*1*))

proposition
fixes *p* :: ‹(*nat*, *nat*) *form*›
shows ‹([] ⊢ *p*) = *valid p*›
using *completeness correctness*
unfolding *model-def* **by** *fastforce*

corollary ‹∀ *e* (*f*::*nat* ⇒ *nat hterm list* ⇒ *nat hterm*) (*g*::*nat* ⇒ *nat hterm list* ⇒ *bool*).
e,*f*,*g*,*ps* ⊨ *p* ⟹ *ps* ⊢ *p*›
by (*rule completeness*)

References

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