

# Meta-theory of first-order predicate logic

Stefan Berghofer

December 14, 2021

## 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

<b>1</b>	<b>First-Order Logic According to Fitting</b>	<b>2</b>
<b>2</b>	<b>Miscellaneous Utilities</b>	<b>2</b>
<b>3</b>	<b>Terms and formulae</b>	<b>2</b>
3.1	Closed terms and formulae . . . . .	3
3.2	Substitution . . . . .	4
3.3	Parameters . . . . .	5
<b>4</b>	<b>Semantics</b>	<b>7</b>
<b>5</b>	<b>Proof calculus</b>	<b>9</b>
<b>6</b>	<b>Correctness</b>	<b>12</b>
<b>7</b>	<b>Completeness</b>	<b>13</b>
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

<b>8</b>	<b>Löwenheim-Skolem theorem</b>	<b>54</b>
<b>9</b>	<b>Completeness for open formulas</b>	<b>59</b>
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]*:  $\langle - A \cap B = B - A \rangle$  **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  $\Rightarrow$  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: ⟨ $i \leq j$ ⟩

shows ⟨closedt  $i$  ( $t :: 'a$  term)  $\implies$  closedt  $j$   $t$ ⟩

and ⟨closedts  $i$  ( $ts :: 'a$  term list)  $\implies$  closedts  $j$   $ts$ ⟩

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

#### theorem closed-mono: assumes le: ⟨ $i \leq j$ ⟩

shows ⟨closed  $i$   $p \implies$  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**

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

**where**

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

**primrec**

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

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

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

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

$\langle closedt\ 0\ (t::'a\ term) \implies closedt\ 0\ (liftt\ t) \rangle$   
 $\langle closedts\ 0\ (ts::'a\ term\ list) \implies closedts\ 0\ (liftts\ ts) \rangle$   
**by** (induct  $t$  **and**  $ts$  rule:  $closedt.induct\ closedts.induct$ ) *simp-all*

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

**assumes**  $u: \langle closedt\ 0\ u \rangle$   
**shows**  $\langle closedt\ (Suc\ i)\ t \implies closedt\ i\ (t[u/i]) \rangle$   
**and**  $\langle closedts\ (Suc\ i)\ ts \implies closedts\ i\ (ts[u/i]) \rangle$   
**using**  $u\ closedt\ mono(1)$   
**by** (induct  $t$  **and**  $ts$  rule:  $closedt.induct\ closedts.induct$ ) *auto*

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

$\langle closedt\ 0\ t \implies closed\ (Suc\ i)\ p \implies closed\ i\ (p[t/i]) \rangle$

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

**theorem** subst-size-form [simp]:  $\langle \text{size-form } (\text{subst } p \ t \ i) = \text{size-form } p \rangle$   
 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 ::  $\langle 'a \ \text{term} \Rightarrow 'a \ \text{set} \rangle$  **and**  
 paramsts ::  $\langle 'a \ \text{term list} \Rightarrow 'a \ \text{set} \rangle$  **where**  
 $\langle \text{paramst } (\text{Var } n) = \{\} \rangle$   
 $\langle \text{paramst } (\text{App } a \ ts) = \{a\} \cup \text{paramsts } ts \rangle$   
 $\langle \text{paramsts } [] = \{\} \rangle$   
 $\langle \text{paramsts } (t \ # \ ts) = (\text{paramst } t \cup \text{paramsts } ts) \rangle$

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

$\langle \text{params } FF = \{\} \rangle$   
 $\langle \text{params } TT = \{\} \rangle$   
 $\langle \text{params } (\text{Pred } b \ ts) = \text{paramsts } ts \rangle$   
 $\langle \text{params } (\text{And } p \ q) = \text{params } p \cup \text{params } q \rangle$   
 $\langle \text{params } (\text{Or } p \ q) = \text{params } p \cup \text{params } q \rangle$   
 $\langle \text{params } (\text{Impl } p \ q) = \text{params } p \cup \text{params } q \rangle$   
 $\langle \text{params } (\text{Neg } p) = \text{params } p \rangle$   
 $\langle \text{params } (\text{Forall } p) = \text{params } p \rangle$   
 $\langle \text{params } (\text{Exists } p) = \text{params } p \rangle$

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

**primrec**

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

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

$\langle \text{psubst } f \ FF = FF \rangle$   
 $\langle \text{psubst } f \ TT = TT \rangle$   
 $\langle \text{psubst } f \ (\text{Pred } b \ ts) = \text{Pred } b \ (\text{psubstts } f \ ts) \rangle$   
 $\langle \text{psubst } f \ (\text{And } p \ q) = \text{And } (\text{psubst } f \ p) \ (\text{psubst } f \ q) \rangle$   
 $\langle \text{psubst } f \ (\text{Or } p \ q) = \text{Or } (\text{psubst } f \ p) \ (\text{psubst } f \ q) \rangle$

$\langle \text{psubst } f \text{ (Impl } p \text{ } q) = \text{Impl } (\text{psubst } f \text{ } p) \text{ (psubst } f \text{ } q) \rangle$   
 $\langle \text{psubst } f \text{ (Neg } p) = \text{Neg } (\text{psubst } f \text{ } p) \rangle$   
 $\langle \text{psubst } f \text{ (Forall } p) = \text{Forall } (\text{psubst } f \text{ } p) \rangle$   
 $\langle \text{psubst } f \text{ (Exists } p) = \text{Exists } (\text{psubst } f \text{ } p) \rangle$

**theorem** *psubstt-closed* [*simp*]:  
 $\langle \text{closedt } i \text{ (psubstt } f \text{ } t) = \text{closedt } i \text{ } t \rangle$   
 $\langle \text{closedts } i \text{ (psubstts } f \text{ } ts) = \text{closedts } i \text{ } ts \rangle$   
**by** (*induct t and ts rule: closedt.induct closedts.induct*) *simp-all*

**theorem** *psubst-closed* [*simp*]:  
 $\langle \text{closed } i \text{ (psubst } f \text{ } p) = \text{closed } i \text{ } p \rangle$   
**by** (*induct p arbitrary: i*) *simp-all*

**theorem** *psubstt-subst* [*simp*]:  
 $\langle \text{psubstt } f \text{ (substt } t \text{ } u \text{ } i) = \text{substt } (\text{psubstt } f \text{ } t) \text{ (psubstt } f \text{ } u) \text{ } i \rangle$   
 $\langle \text{psubstts } f \text{ (substts } ts \text{ } u \text{ } i) = \text{substts } (\text{psubstts } f \text{ } ts) \text{ (psubstt } f \text{ } u) \text{ } i \rangle$   
**by** (*induct t and ts rule: psubstt.induct psubstts.induct*) *simp-all*

**theorem** *psubstt-lift* [*simp*]:  
 $\langle \text{psubstt } f \text{ (liftt } t) = \text{liftt } (\text{psubstt } f \text{ } t) \rangle$   
 $\langle \text{psubstts } f \text{ (liftts } ts) = \text{liftts } (\text{psubstts } f \text{ } ts) \rangle$   
**by** (*induct t and ts rule: psubstt.induct psubstts.induct*) *simp-all*

**theorem** *psubst-subst* [*simp*]:  
 $\langle \text{psubst } f \text{ (subst } P \text{ } t \text{ } i) = \text{subst } (\text{psubst } f \text{ } P) \text{ (psubstt } f \text{ } t) \text{ } i \rangle$   
**by** (*induct P arbitrary: i t*) *simp-all*

**theorem** *psubstt-upd* [*simp*]:  
 $\langle x \notin \text{paramst } (t :: 'a \text{ term}) \implies \text{psubstt } (f(x := y)) \text{ } t = \text{psubstt } f \text{ } t \rangle$   
 $\langle x \notin \text{paramsts } (ts :: 'a \text{ term list}) \implies \text{psubstts } (f(x := y)) \text{ } ts = \text{psubstts } f \text{ } ts \rangle$   
**by** (*induct t and ts rule: psubstt.induct psubstts.induct (auto split: sum.split)*)

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

**theorem** *psubstt-id*:  
**fixes**  $t :: \langle 'a \text{ term} \rangle$  **and**  $ts :: \langle 'a \text{ term list} \rangle$   
**shows**  $\langle \text{psubstt id } t = t \rangle$  **and**  $\langle \text{psubstts } (\lambda x. x) \text{ } ts = ts \rangle$   
**by** (*induct t and ts rule: psubstt.induct psubstts.induct*) *simp-all*

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

**proof**

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

**show**  $\langle \text{psubst id } p = \text{id } p \rangle$

**by** (*induct p*) (*simp-all add: psubstt-id*)

**qed**

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

$\langle \text{paramst } (\text{psubstt } f \ t) = f \ \langle \text{paramst } t \rangle$

$\langle \text{paramsts } (\text{psubstts } f \ ts) = f \ \langle \text{paramsts } ts \rangle$

**by** (*induct t and ts rule: paramst.induct paramsts.induct*) (*simp-all add: image-Un*)

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

**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* ::  $\langle (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a \rangle$  (*-(-:-) [90, 0, 0] 91*)

**where**

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

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

**by** (*simp add: shift-def*)

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

**by** (*simp add: shift-def*)

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

**by** (*simp add: shift-def*)

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

**proof**

**fix**  $x$

**show**  $\langle (e\langle i:U \rangle \langle 0:T \rangle) \ x = (e\langle 0:T \rangle \langle \text{Suc } i:U \rangle) \ x \rangle$

**by** (*cases x*) (*simp-all add: shift-def*)

**qed**

**primrec**

*evalt* ::  $\langle (\text{nat} \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \ \text{list} \Rightarrow 'c) \Rightarrow 'a \ \text{term} \Rightarrow 'c \rangle$  **and**

*evalts* ::  $\langle (\text{nat} \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \ \text{list} \Rightarrow 'c) \Rightarrow 'a \ \text{term list} \Rightarrow 'c \ \text{list} \rangle$  **where**

$\langle \text{evalt } e \ f \ (\text{Var } n) = e \ n \rangle$

|  $\langle \text{evalt } e \ f \ (\text{App } a \ ts) = f \ a \ (\text{evalts } e \ f \ ts) \rangle$

|  $\langle \text{evalts } e \ f \ [] = [] \rangle$

|  $\langle \text{evalts } e \ f \ (t \ # \ ts) = \text{evalt } e \ f \ t \ # \ \text{evalts } e \ f \ ts \rangle$

**primrec** *eval* ::  $\langle (\text{nat} \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c \ \text{list} \Rightarrow 'c) \Rightarrow$

$('b \Rightarrow 'c \ \text{list} \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \ \text{form} \Rightarrow \text{bool} \rangle$  **where**

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

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

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

The following substitution lemmas relate substitution and evaluation functions:

**theorem**  $\text{subst-lemma}' [simp]$ :  
 $\langle \text{evalt } e \text{ f } (\text{substt } t \ u \ i) = \text{evalt } (e\langle i: \text{evalt } e \text{ f } u \rangle) \text{ f } t \rangle$   
 $\langle \text{evalts } e \text{ f } (\text{substts } ts \ u \ i) = \text{evalts } (e\langle i: \text{evalt } e \text{ f } u \rangle) \text{ f } ts \rangle$   
**by**  $(\text{induct } t \ \mathbf{and} \ ts \ \text{rule: } \text{substt.induct } \text{substts.induct}) \ \text{simp-all}$

**theorem**  $\text{lift-lemma} [simp]$ :  
 $\langle \text{evalt } (e\langle 0:z \rangle) \text{ f } (\text{liftt } t) = \text{evalt } e \text{ f } t \rangle$   
 $\langle \text{evalts } (e\langle 0:z \rangle) \text{ f } (\text{liftts } ts) = \text{evalts } e \text{ f } ts \rangle$   
**by**  $(\text{induct } t \ \mathbf{and} \ ts \ \text{rule: } \text{liftt.induct } \text{liftts.induct}) \ \text{simp-all}$

**theorem**  $\text{subst-lemma} [simp]$ :  
 $\langle \text{eval } e \text{ f g } (\text{subst } a \ t \ i) = \text{eval } (e\langle i: \text{evalt } e \text{ f } t \rangle) \text{ f g } a \rangle$   
**by**  $(\text{induct } a \ \text{arbitrary: } e \ i \ t) \ \text{simp-all}$

**theorem**  $\text{upd-lemma}' [simp]$ :  
 $\langle n \notin \text{params } t \Longrightarrow \text{evalt } e \ (f(n := x)) \ t = \text{evalt } e \text{ f } t \rangle$   
 $\langle n \notin \text{paramsts } ts \Longrightarrow \text{evalts } e \ (f(n := x)) \ ts = \text{evalts } e \text{ f } ts \rangle$   
**by**  $(\text{induct } t \ \mathbf{and} \ ts \ \text{rule: } \text{paramst.induct } \text{paramsts.induct}) \ \text{auto}$

**theorem**  $\text{upd-lemma} [simp]$ :  
 $\langle n \notin \text{params } p \Longrightarrow \text{eval } e \ (f(n := x)) \ g \ p = \text{eval } e \text{ f g } p \rangle$   
**by**  $(\text{induct } p \ \text{arbitrary: } e) \ \text{simp-all}$

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

**theorem**  $\text{psubst-eval}' [simp]$ :  
 $\langle \text{evalt } e \text{ f } (\text{psubstt } h \ t) = \text{evalt } e \ (\lambda p. \text{f } (h \ p)) \ t \rangle$   
 $\langle \text{evalts } e \text{ f } (\text{psubstts } h \ ts) = \text{evalts } e \ (\lambda p. \text{f } (h \ p)) \ ts \rangle$



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

**theorem**  $psubst-eval$ :

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

by (induct  $p$  arbitrary:  $e$ )  $simp-all$

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

**theorem**  $ex-all-commute-eval$ :

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

$(Forall\ (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1]))) \rangle$

apply  $simp$

Simplification yields the following proof state:

1.  $(\exists z. \forall za. g\ p\ [z, za]) \longrightarrow (\forall z. \exists za. g\ p\ [za, z])$

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**  $deriv :: \langle ('a, 'b)\ form\ list \Rightarrow ('a, 'b)\ form \Rightarrow bool \rangle (- \vdash - [50,50] 50)$

**where**

$Assum: \langle a \in set\ G \Longrightarrow G \vdash a \rangle$

|  $TTI: \langle G \vdash TT \rangle$

|  $FFE: \langle G \vdash FF \Longrightarrow G \vdash a \rangle$

|  $NegI: \langle a \# G \vdash FF \Longrightarrow G \vdash Neg\ a \rangle$

|  $NegE: \langle G \vdash Neg\ a \Longrightarrow G \vdash a \Longrightarrow G \vdash FF \rangle$

|  $Class: \langle Neg\ a \# G \vdash FF \Longrightarrow G \vdash a \rangle$

|  $AndI: \langle G \vdash a \Longrightarrow G \vdash b \Longrightarrow G \vdash And\ a\ b \rangle$

|  $AndE1: \langle G \vdash And\ a\ b \Longrightarrow G \vdash a \rangle$

|  $AndE2: \langle G \vdash And\ a\ b \Longrightarrow G \vdash b \rangle$

|  $OrI1: \langle G \vdash a \Longrightarrow G \vdash Or\ a\ b \rangle$

|  $OrI2: \langle G \vdash b \Longrightarrow G \vdash Or\ a\ b \rangle$

|  $OrE: \langle G \vdash Or\ a\ b \Longrightarrow a \# G \vdash c \Longrightarrow b \# G \vdash c \Longrightarrow G \vdash c \rangle$

|  $ImplI: \langle a \# G \vdash b \Longrightarrow G \vdash Impl\ a\ b \rangle$

|  $ImplE: \langle G \vdash Impl\ a\ b \Longrightarrow G \vdash a \Longrightarrow G \vdash b \rangle$

|  $ForallI: \langle G \vdash a[App\ n\ []/0] \Longrightarrow list-all\ (\lambda p. n \notin params\ p)\ G \Longrightarrow n \notin params\ a \Longrightarrow G \vdash Forall\ a \rangle$

|  $ForallE: \langle G \vdash Forall\ a \Longrightarrow G \vdash a[t/0] \rangle$

|  $ExistsI: \langle G \vdash a[t/0] \Longrightarrow G \vdash Exists\ a \rangle$

|  $ExistsE: \langle G \vdash Exists\ a \Longrightarrow a[App\ n\ []/0] \# G \vdash b \Longrightarrow$

$list-all\ (\lambda p. n \notin params\ p)\ G \Longrightarrow n \notin params\ a \Longrightarrow n \notin params\ b \Longrightarrow G \vdash b \rangle$

The following derived inference rules are sometimes useful in applications.

**theorem** *Class'*:  $\langle \text{Neg } A \# G \vdash A \implies G \vdash A \rangle$   
 by (*rule Class*, *rule NegE*, *rule Assum*) (*simp*, *iprover*)

**theorem** *cut*:  $\langle G \vdash A \implies A \# G \vdash B \implies G \vdash B \rangle$   
 by (*rule ImplE*, *rule ImplI*)

**theorem** *ForallE'*:  $\langle G \vdash \text{Forall } a \implies \text{subst } a \text{ } t \ 0 \# G \vdash B \implies G \vdash B \rangle$   
 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*:  $\langle [] \vdash \text{Or } (\text{Pred } p \ []) (\text{Neg } (\text{Pred } p \ [])) \rangle$  (**is**  $\langle - \vdash ?or \rangle$ )

**proof** –

have  $\langle [\text{Neg } ?or] \vdash \text{Neg } ?or \rangle$   
 by (*simp add: Assum*)  
 moreover { have  $\langle [\text{Pred } p \ [], \text{Neg } ?or] \vdash \text{Neg } ?or \rangle$   
 by (*simp add: Assum*)  
 moreover have  $\langle [\text{Pred } p \ [], \text{Neg } ?or] \vdash \text{Pred } p \ [] \rangle$   
 by (*simp add: Assum*)  
 then have  $\langle [\text{Pred } p \ [], \text{Neg } ?or] \vdash ?or \rangle$   
 by (*rule OrI1*)  
 ultimately have  $\langle [\text{Pred } p \ [], \text{Neg } ?or] \vdash FF \rangle$   
 by (*rule NegE*)  
 then have  $\langle [\text{Neg } ?or] \vdash \text{Neg } (\text{Pred } p \ []) \rangle$   
 by (*rule NegI*)  
 then have  $\langle [\text{Neg } ?or] \vdash ?or \rangle$   
 by (*rule OrI2*) }  
 ultimately have  $\langle [\text{Neg } ?or] \vdash FF \rangle$   
 by (*rule NegE*)  
 then show *?thesis*  
 by (*rule Class*)

qed

**theorem** *ex-all-commute*:

$\langle ([::(\text{nat}, 'b) \text{ form list}) \vdash \text{Impl } (\text{Exists } (\text{Forall } (\text{Pred } p \ [\text{Var } 1, \text{Var } 0])))$   
 $(\text{Forall } (\text{Exists } (\text{Pred } p \ [\text{Var } 0, \text{Var } 1]))) \rangle$

**proof** –

let *?forall* =  $\langle \text{Forall } (\text{Pred } p \ [\text{Var } 1, \text{Var } 0]) :: (\text{nat}, 'b) \text{ form} \rangle$

have  $\langle [\text{Exists } ?forall] \vdash \text{Exists } ?forall \rangle$

by (*simp add: Assum*)

moreover { have  $\langle [?forall[\text{App } 1 \ []/0], \text{Exists } ?forall] \vdash \text{Forall } (\text{Pred } p \ [\text{App } 1 \ []/0], \text{Var } 0]) \rangle$

by (*simp add: Assum*)

moreover have  $\langle [\text{Pred } p \ [\text{App } 1 \ [], \text{Var } 0][\text{App } 0 \ []/0], ?forall[\text{App } 1 \ []/0],$

$\text{Exists } ?forall] \vdash \text{Pred } p \ [\text{Var } 0, \text{App } 0 \ []][\text{App } 1 \ []/0] \rangle$

by (*simp add: Assum*)

ultimately have  $\langle [?forall[\text{App } 1 \ []/0], \text{Exists } ?forall] \vdash (\text{Pred } p \ [\text{Var } 0, \text{App } 0$

$\square$ ) $[App\ 1\ \square/0]$   
**by** (rule *ForallE'*) }  
**then have**  $\langle [?forall[App\ 1\ \square/0],\ Exists\ ?forall] \vdash Exists\ (Pred\ p\ [Var\ 0,\ App\ 0\ \square]) \rangle$   
**by** (rule *ExistsI*)  
**moreover have**  $\langle list\text{-}all\ (\lambda p.\ 1 \notin params\ p)\ [Exists\ ?forall] \rangle$   
**by** *simp*  
**moreover have**  $\langle 1 \notin params\ ?forall \rangle$   
**by** *simp*  
**moreover have**  $\langle 1 \notin params\ (Exists\ (Pred\ p\ [Var\ 0,\ App\ (0 :: nat)\ \square])) \rangle$   
**by** *simp*  
**ultimately have**  $\langle [Exists\ ?forall] \vdash Exists\ (Pred\ p\ [Var\ 0,\ App\ 0\ \square]) \rangle$   
**by** (rule *ExistsE*)  
**then have**  $\langle [Exists\ ?forall] \vdash (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1]))[App\ 0\ \square/0] \rangle$   
**by** *simp*  
**moreover have**  $\langle list\text{-}all\ (\lambda p.\ 0 \notin params\ p)\ [Exists\ ?forall] \rangle$   
**by** *simp*  
**moreover have**  $\langle 0 \notin params\ (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1])) \rangle$   
**by** *simp*  
**ultimately have**  $\langle [Exists\ ?forall] \vdash Forall\ (Exists\ (Pred\ p\ [Var\ 0,\ Var\ 1])) \rangle$   
**by** (rule *ForallI*)  
**then show** *?thesis*  
**by** (rule *ImplI*)  
**qed**

**theorem** *drinker*:  $\langle (\square :: (nat,\ 'b)\ form\ list) \vdash Exists\ (Impl\ (Pred\ P\ [Var\ 0])\ (Forall\ (Pred\ P\ [Var\ 0]))) \rangle$   
**proof** –  
**let** *?impl* =  $\langle (Impl\ (Pred\ P\ [Var\ 0])\ (Forall\ (Pred\ P\ [Var\ 0]))) :: (nat,\ 'b)\ form \rangle$   
**let** *?G'* =  $\langle [Pred\ P\ [Var\ 0],\ Neg\ (Exists\ ?impl)] \rangle$   
**let** *?G* =  $\langle Neg\ (Pred\ P\ [App\ 0\ \square]) \# ?G' \rangle$   
  
**have**  $\langle ?G \vdash Neg\ (Exists\ ?impl) \rangle$   
**by** (*simp add: Assum*)  
**moreover have**  $\langle Pred\ P\ [App\ 0\ \square] \# ?G \vdash Neg\ (Pred\ P\ [App\ 0\ \square]) \rangle$   
**and**  $\langle Pred\ P\ [App\ 0\ \square] \# ?G \vdash Pred\ P\ [App\ 0\ \square] \rangle$   
**by** (*simp-all add: Assum*)  
**then have**  $\langle Pred\ P\ [App\ 0\ \square] \# ?G \vdash FF \rangle$   
**by** (rule *NegE*)  
**then have**  $\langle Pred\ P\ [App\ 0\ \square] \# ?G \vdash Forall\ (Pred\ P\ [Var\ 0]) \rangle$   
**by** (rule *FFE*)  
**then have**  $\langle ?G \vdash ?impl[App\ 0\ \square/0] \rangle$   
**using** *ImplI* **by** *simp*  
**then have**  $\langle ?G \vdash Exists\ ?impl \rangle$   
**by** (rule *ExistsI*)  
**ultimately have**  $\langle ?G \vdash FF \rangle$   
**by** (rule *NegE*)  
**then have**  $\langle ?G' \vdash Pred\ P\ [Var\ 0][App\ 0\ \square/0] \rangle$   
**using** *Class* **by** *simp*

**moreover have**  $\langle \text{list-all } (\lambda p. (0 :: \text{nat}) \notin \text{params } p) \text{ ?}G' \rangle$   
**by** *simp*  
**moreover have**  $\langle (0 :: \text{nat}) \notin \text{params } (\text{Pred } P \text{ [Var } 0]) \rangle$   
**by** *simp*  
**ultimately have**  $\langle \text{?}G' \vdash \text{Forall } (\text{Pred } P \text{ [Var } 0]) \rangle$   
**by** (*rule ForallI*)  
**then have**  $\langle [\text{Neg } (\text{Exists } \text{?impl})] \vdash \text{?impl}[\text{Var } 0/0] \rangle$   
**using** *ImplI* **by** *simp*  
**then have**  $\langle [\text{Neg } (\text{Exists } \text{?impl})] \vdash \text{Exists } \text{?impl} \rangle$   
**by** (*rule ExistsI*)  
**then show** *?thesis*  
**by** (*rule Class'*)  
**qed**

**theorem** *peirce*:

$\langle [] \vdash \text{Impl } (\text{Impl } (\text{Impl } (\text{Pred } P \ [])) (\text{Pred } Q \ [])) (\text{Pred } P \ []) (\text{Pred } P \ []) \rangle$   
**(is**  $\langle [] \vdash \text{Impl } \text{?PQP } (\text{Pred } P \ []) \rangle$ )

**proof** –

**let**  $\text{?PQPP} = \langle \text{Impl } \text{?PQP } (\text{Pred } P \ []) \rangle$

**have**  $\langle [\text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{?PQP} \rangle$

**by** (*simp add: Assum*)

**moreover { have**  $\langle [\text{Pred } P \ [], \text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{Neg } \text{?PQPP} \rangle$

**by** (*simp add: Assum*)

**moreover have**  $\langle [\text{?PQP}, \text{Pred } P \ [], \text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{Pred } P \ [] \rangle$

**by** (*simp add: Assum*)

**then have**  $\langle [\text{Pred } P \ [], \text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{?PQPP} \rangle$

**by** (*rule ImplI*)

**ultimately have**  $\langle [\text{Pred } P \ [], \text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{FF} \rangle$

**by** (*rule NegE*) }

**then have**  $\langle [\text{Pred } P \ [], \text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{Pred } Q \ [] \rangle$

**by** (*rule FFE*)

**then have**  $\langle [\text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{Impl } (\text{Pred } P \ []) (\text{Pred } Q \ []) \rangle$

**by** (*rule ImplI*)

**ultimately have**  $\langle [\text{?PQP}, \text{Neg } \text{?PQPP}] \vdash \text{Pred } P \ [] \rangle$

**by** (*rule ImplE*)

**then have**  $\langle [\text{Neg } \text{?PQPP}] \vdash \text{?PQPP} \rangle$

**by** (*rule ImplI*)

**then show**  $\langle [] \vdash \text{?PQPP} \rangle$

**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*:  $\langle G \vdash p \implies \forall e f g. e, f, g, G \models p \rangle$

```

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

The latter property is called the *model existence theorem*. To see why 2 implies 1, observe that  $Neg p, ps \not\vdash FF$  implies that  $Neg p, ps$  is consistent, which, by the model existence theorem, implies that  $Neg p, ps$  has a model, which in turn implies that  $ps \not\models p$ . By contraposition, it therefore follows from  $ps \models p$  that  $Neg p, ps \vdash FF$ , which allows us to deduce  $ps \vdash p$  using rule *Class*.

In most textbooks on logic, a set  $S$  of formulae is called *consistent*, if no contradiction can be derived from  $S$  using a *specific proof calculus*, i.e.  $S \not\vdash FF$ . Rather than defining consistency relative to a *specific* calculus, Fitting

uses the more general approach of describing properties that all consistent sets must have (see §7.1).

The key idea behind the proof of the model existence theorem is to extend a consistent set to one that is *maximal* (see §7.5). In order to do this, we use the fact that the set of formulae is enumerable (see §7.4), which allows us to form a sequence  $\phi_0, \phi_1, \phi_2, \dots$  containing all formulae. We can then construct a sequence  $S_i$  of consistent sets as follows:

$$S_0 = S$$

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

To obtain a maximal consistent set, we form the union  $\bigcup_i S_i$  of these sets. To ensure that this union is still consistent, additional closure (see §7.2) and finiteness (see §7.3) properties are needed. It can be shown that a maximal consistent set is a *Hintikka set* (see §7.6). Hintikka sets are satisfiable in *Herbrand* models, where closed terms coincide with their interpretation.

## 7.1 Consistent sets

In this section, we describe an abstract criterion for consistent sets. A set of sets of formulae is called a *consistency property*, if the following holds:

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

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

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

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

$$(\forall Z. \text{Neg } (\text{Neg } Z) \in S \longrightarrow S \cup \{Z\} \in C) \wedge$$

$$(\forall A B. \text{And } A B \in S \longrightarrow S \cup \{A, B\} \in C) \wedge$$

$$(\forall A B. \text{Neg } (\text{Or } A B) \in S \longrightarrow S \cup \{\text{Neg } A, \text{Neg } B\} \in C) \wedge$$

$$(\forall A B. \text{Or } A B \in S \longrightarrow S \cup \{A\} \in C \vee S \cup \{B\} \in C) \wedge$$

$$(\forall A B. \text{Neg } (\text{And } A B) \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{\text{Neg } B\} \in C) \wedge$$

$$(\forall A B. \text{Impl } A B \in S \longrightarrow S \cup \{\text{Neg } A\} \in C \vee S \cup \{B\} \in C) \wedge$$

$$(\forall A B. \text{Neg } (\text{Impl } A B) \in S \longrightarrow S \cup \{A, \text{Neg } B\} \in C) \wedge$$

$$(\forall P t. \text{closedt } 0 t \longrightarrow \text{Forall } P \in S \longrightarrow S \cup \{P[t/0]\} \in C) \wedge$$

$$(\forall P t. \text{closedt } 0 t \longrightarrow \text{Neg } (\text{Exists } P) \in S \longrightarrow S \cup \{\text{Neg } (P[t/0])\} \in C) \wedge$$

$$(\forall P. \text{Exists } P \in S \longrightarrow (\exists x. S \cup \{P[\text{App } x \ \_ / 0]\} \in C)) \wedge$$

$$(\forall P. \text{Neg } (\text{Forall } P) \in S \longrightarrow (\exists x. S \cup \{\text{Neg } (P[\text{App } x \ \_ / 0])\} \in C)) \rangle$$

In §7.3, we will show how to extend a consistency property to one that is of *finite character*. However, the above definition of a consistency property cannot be used for this, since there is a problem with the treatment of formulae of the form *Exists P* and *Neg (Forall P)*. Fitting therefore suggests to define an *alternative consistency property* as follows:

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

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

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

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

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

**definition** *mk-alt-consistency* ::  $\langle 'a, 'b \text{ form set set} \Rightarrow ('a, 'b \text{ form set set}) \rangle$   
**where**

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

**theorem** *alt-consistency*:

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

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

**unfolding** *alt-consistency-def*

**proof** (*intro allI impI conjI*)

**fix**  $f :: \langle 'a \Rightarrow 'a \rangle$  **and**  $S :: \langle ('a, 'b \text{ form set}) \rangle$

**assume**  $\langle S \in \text{mk-alt-consistency } C \rangle$

**then obtain**  $f$  **where**  $sc: \langle \text{psubst } f \text{ ' } S \in C \rangle$  (**is**  $\langle ?S' \in C \rangle$ )

**unfolding** *mk-alt-consistency-def* **by** *blast*

**fix**  $p \ ts$

**show**  $\langle \neg ( \text{Pred } p \ ts \in S \wedge \text{Neg } (\text{Pred } p \ ts) \in S) \rangle$

**proof**

**assume**  $*$ :  $\langle \text{Pred } p \ ts \in S \wedge \text{Neg } (\text{Pred } p \ ts) \in S \rangle$

**then have**  $\langle \text{psubst } f \ (\text{Pred } p \ ts) \in ?S' \rangle$

**by** *blast*

**then have**  $\langle \text{Pred } p \ (\text{psubstts } f \ ts) \in ?S' \rangle$

**by** *simp*

**then have**  $\langle \text{Neg } (\text{Pred } p \ (\text{psubstts } f \ ts)) \notin ?S' \rangle$

**using** *conc sc* **by** (*simp add: consistency-def*)

**then have**  $\langle \text{Neg } (\text{Pred } p \ ts) \notin S \rangle$

**by** *force*

**then show** *False*

**using**  $*$  **by** *blast*

qed

have  $\langle FF \notin ?S' \rangle$  and  $\langle Neg TT \notin ?S' \rangle$   
using *conc sc unfolding consistency-def* by *simp-all*  
then show  $\langle FF \notin S \rangle$  and  $\langle Neg TT \notin S \rangle$   
by (*force, force*)

```
{ fix Z
  assume  $\langle Neg (Neg Z) \in S \rangle$ 
  then have  $\langle psubst f (Neg (Neg Z)) \in ?S' \rangle$ 
    by blast
  then have  $\langle ?S' \cup \{psubst f Z\} \in C \rangle$ 
    using conc sc by (simp add: consistency-def)
  then show  $\langle S \cup \{Z\} \in ?C' \rangle$ 
    unfolding mk-alt-consistency-def by auto }
```

```
{ fix A B
  assume  $\langle And A B \in S \rangle$ 
  then have  $\langle psubst f (And A B) \in ?S' \rangle$ 
    by blast
  then have  $\langle ?S' \cup \{psubst f A, psubst f B\} \in C \rangle$ 
    using conc sc by (simp add: consistency-def)
  then show  $\langle S \cup \{A, B\} \in ?C' \rangle$ 
    unfolding mk-alt-consistency-def by auto }
```

```
{ fix A B
  assume  $\langle Neg (Or A B) \in S \rangle$ 
  then have  $\langle psubst f (Neg (Or A B)) \in ?S' \rangle$ 
    by blast
  then have  $\langle ?S' \cup \{Neg (psubst f A), Neg (psubst f B)\} \in C \rangle$ 
    using conc sc by (simp add: consistency-def)
  then show  $\langle S \cup \{Neg A, Neg B\} \in ?C' \rangle$ 
    unfolding mk-alt-consistency-def by auto }
```

```
{ fix A B
  assume  $\langle Neg (Impl A B) \in S \rangle$ 
  then have  $\langle psubst f (Neg (Impl A B)) \in ?S' \rangle$ 
    by blast
  then have  $\langle ?S' \cup \{psubst f A, Neg (psubst f B)\} \in C \rangle$ 
    using conc sc by (simp add: consistency-def)
  then show  $\langle S \cup \{A, Neg B\} \in ?C' \rangle$ 
    unfolding mk-alt-consistency-def by auto }
```

```
{ fix A B
  assume  $\langle Or A B \in S \rangle$ 
  then have  $\langle psubst f (Or A B) \in ?S' \rangle$ 
    by blast
  then have  $\langle ?S' \cup \{psubst f A\} \in C \vee ?S' \cup \{psubst f B\} \in C \rangle$ 
    using conc sc by (simp add: consistency-def)
```



**then show**  $\langle S \cup \{A\} \in ?C' \vee S \cup \{B\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *auto* }

{ **fix**  $A B$   
**assume**  $\langle \text{Neg } (And A B) \in S \rangle$   
**then have**  $\langle \text{psubst } f (\text{Neg } (And A B)) \in ?S' \rangle$   
**by** *blast*  
**then have**  $\langle ?S' \cup \{\text{Neg } (\text{psubst } f A)\} \in C \vee ?S' \cup \{\text{Neg } (\text{psubst } f B)\} \in C \rangle$   
**using** *conc sc* **by** (*simp add: consistency-def*)  
**then show**  $\langle S \cup \{\text{Neg } A\} \in ?C' \vee S \cup \{\text{Neg } B\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *auto* }

{ **fix**  $A B$   
**assume**  $\langle \text{Impl } A B \in S \rangle$   
**then have**  $\langle \text{psubst } f (\text{Impl } A B) \in ?S' \rangle$   
**by** *blast*  
**then have**  $\langle ?S' \cup \{\text{Neg } (\text{psubst } f A)\} \in C \vee ?S' \cup \{\text{psubst } f B\} \in C \rangle$   
**using** *conc sc* **by** (*simp add: consistency-def*)  
**then show**  $\langle S \cup \{\text{Neg } A\} \in ?C' \vee S \cup \{B\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *auto* }

{ **fix**  $P$  **and**  $t :: \langle 'a \text{ term} \rangle$   
**assume**  $\langle \text{closedt } 0 t \rangle$  **and**  $\langle \text{Forall } P \in S \rangle$   
**then have**  $\langle \text{psubst } f (\text{Forall } P) \in ?S' \rangle$   
**by** *blast*  
**then have**  $\langle ?S' \cup \{\text{psubst } f P[\text{psubstt } f t/0]\} \in C \rangle$   
**using**  $\langle \text{closedt } 0 t \rangle$  *conc sc* **by** (*simp add: consistency-def*)  
**then show**  $\langle S \cup \{P[t/0]\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *auto* }

{ **fix**  $P$  **and**  $t :: \langle 'a \text{ term} \rangle$   
**assume**  $\langle \text{closedt } 0 t \rangle$  **and**  $\langle \text{Neg } (\text{Exists } P) \in S \rangle$   
**then have**  $\langle \text{psubst } f (\text{Neg } (\text{Exists } P)) \in ?S' \rangle$   
**by** *blast*  
**then have**  $\langle ?S' \cup \{\text{Neg } (\text{psubst } f P[\text{psubstt } f t/0])\} \in C \rangle$   
**using**  $\langle \text{closedt } 0 t \rangle$  *conc sc* **by** (*simp add: consistency-def*)  
**then show**  $\langle S \cup \{\text{Neg } (P[t/0])\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *auto* }

{ **fix**  $P :: \langle ('a, 'b) \text{ form} \rangle$  **and**  $x f'$   
**assume**  $\langle \forall a \in S. x \notin \text{params } a \rangle$  **and**  $\langle \text{Exists } P \in S \rangle$   
**moreover have**  $\langle \text{psubst } f (\text{Exists } P) \in ?S' \rangle$   
**using** *calculation* **by** *blast*  
**then have**  $\langle \exists y. ?S' \cup \{\text{psubst } f P[\text{App } y \ \_ / 0]\} \in C \rangle$   
**using** *conc sc* **by** (*simp add: consistency-def*)  
**then obtain**  $y$  **where**  $\langle ?S' \cup \{\text{psubst } f P[\text{App } y \ \_ / 0]\} \in C \rangle$   
**by** *blast*

**moreover have**  $\langle \text{psubst } (f(x := y)) \ 'S = ?S' \rangle$

**using** *calculation* **by** (*simp cong add: image-cong*)  
**moreover have**  $\langle \text{psubst } (f(x := y)) \text{ '}$   
 $S \cup \{\text{psubst } (f(x := y)) P[\text{App } ((f(x := y)) x) []/0]\} \in C \rangle$   
**using** *calculation* **by** *auto*  
**ultimately have**  $\langle \exists f. \text{psubst } f \text{ ' } S \cup \{\text{psubst } f P[\text{App } (f x) []/0]\} \in C \rangle$   
**by** *blast*  
**then show**  $\langle S \cup \{P[\text{App } x []/0]\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *simp* **}**

**{ fix**  $P :: \langle ('a, 'b) \text{ form} \rangle$  **and**  $x$   
**assume**  $\langle \forall a \in S. x \notin \text{params } a \rangle$  **and**  $\langle \text{Neg } (\text{Forall } P) \in S \rangle$   
**moreover have**  $\langle \text{psubst } f (\text{Neg } (\text{Forall } P)) \in ?S' \rangle$   
**using** *calculation* **by** *blast*  
**then have**  $\langle \exists y. ?S' \cup \{\text{Neg } (\text{psubst } f P[\text{App } y []/0])\} \in C \rangle$   
**using** *conc sc* **by** (*simp add: consistency-def*)  
**then obtain**  $y$  **where**  $\langle ?S' \cup \{\text{Neg } (\text{psubst } f P[\text{App } y []/0])\} \in C \rangle$   
**by** *blast*

**moreover have**  $\langle \text{psubst } (f(x := y)) \text{ ' } S = ?S' \rangle$   
**using** *calculation* **by** (*simp cong add: image-cong*)  
**moreover have**  $\langle \text{psubst } (f(x := y)) \text{ '}$   
 $S \cup \{\text{Neg } (\text{psubst } (f(x := y)) P[\text{App } ((f(x := y)) x) []/0])\} \in C \rangle$   
**using** *calculation* **by** *auto*  
**ultimately have**  $\langle \exists f. \text{psubst } f \text{ ' } S \cup \{\text{Neg } (\text{psubst } f P[\text{App } (f x) []/0])\} \in C \rangle$   
**by** *blast*  
**then show**  $\langle S \cup \{\text{Neg } (P[\text{App } x []/0])\} \in ?C' \rangle$   
**unfolding** *mk-alt-consistency-def* **by** *simp* **}**

qed

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

**unfolding** *mk-alt-consistency-def*

**proof**

**fix**  $x$  **assume**  $\langle x \in C \rangle$   
**then have**  $\langle \text{psubst } \text{id} \text{ ' } x \in C \rangle$   
**by** *simp*  
**then have**  $\langle \exists f. \text{psubst } f \text{ ' } x \in C \rangle$   
**by** *blast*  
**then show**  $\langle x \in \{S. \exists f. \text{psubst } f \text{ ' } S \in C\} \rangle$   
**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* ::  $\langle ('a, 'b) \text{ form set set} \Rightarrow ('a, 'b) \text{ form set set} \rangle$  **where**  
 $\langle \text{close } C = \{S. \exists S' \in C. S \subseteq S'\} \rangle$

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

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

**lemma** *subset-in-close*:

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

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

**proof** –

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

**unfolding** *close-def* **using**  $\langle S \subseteq S' \rangle$  **by** *blast*

**then show** *?thesis* **unfolding** *close-def* **by** *blast*

**qed**

**theorem** *close-consistency*:

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

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

**unfolding** *consistency-def*

**proof** (*intro allI impI conjI*)

**fix** *S*

**assume**  $\langle S \in \text{close } C \rangle$

**then obtain** *x* **where**  $\langle x \in C \rangle$  **and**  $\langle S \subseteq x \rangle$

**unfolding** *close-def* **by** *blast*

{ **fix** *p ts*

**have**  $\langle \neg (Pred\ p\ ts \in x \wedge Neg\ (Pred\ p\ ts) \in x) \rangle$

**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*

**then show**  $\langle \neg (Pred\ p\ ts \in S \wedge Neg\ (Pred\ p\ ts) \in S) \rangle$

**using**  $\langle S \subseteq x \rangle$  **by** *blast* }

{ **have**  $\langle FF \notin x \rangle$

**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *blast*

**then show**  $\langle FF \notin S \rangle$

**using**  $\langle S \subseteq x \rangle$  **by** *blast* }

{ **have**  $\langle Neg\ TT \notin x \rangle$

**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *blast*

**then show**  $\langle Neg\ TT \notin S \rangle$

**using**  $\langle S \subseteq x \rangle$  **by** *blast* }

{ **fix** *Z*

**assume**  $\langle Neg\ (Neg\ Z) \in S \rangle$

**then have**  $\langle Neg\ (Neg\ Z) \in x \rangle$

**using**  $\langle S \subseteq x \rangle$  **by** *blast*

**then have**  $\langle x \cup \{Z\} \in C \rangle$

**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*

**then show**  $\langle S \cup \{Z\} \in \text{close } C \rangle$

**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

{ **fix** *A B*

**assume**  $\langle And\ A\ B \in S \rangle$

**then have**  $\langle And\ A\ B \in x \rangle$

**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{A, B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*  
**then show**  $\langle S \cup \{A, B\} \in \text{close } C \rangle$   
**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

**{ fix**  $A B$   
**assume**  $\langle \text{Neg } (Or A B) \in S \rangle$   
**then have**  $\langle \text{Neg } (Or A B) \in x \rangle$   
**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{\text{Neg } A, \text{Neg } B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*  
**then show**  $\langle S \cup \{\text{Neg } A, \text{Neg } B\} \in \text{close } C \rangle$   
**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

**{ fix**  $A B$   
**assume**  $\langle Or A B \in S \rangle$   
**then have**  $\langle Or A B \in x \rangle$   
**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{A\} \in C \vee x \cup \{B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*  
**then show**  $\langle S \cup \{A\} \in \text{close } C \vee S \cup \{B\} \in \text{close } C \rangle$   
**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

**{ fix**  $A B$   
**assume**  $\langle \text{Neg } (And A B) \in S \rangle$   
**then have**  $\langle \text{Neg } (And A B) \in x \rangle$   
**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{\text{Neg } A\} \in C \vee x \cup \{\text{Neg } B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*  
**then show**  $\langle S \cup \{\text{Neg } A\} \in \text{close } C \vee S \cup \{\text{Neg } B\} \in \text{close } C \rangle$   
**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

**{ fix**  $A B$   
**assume**  $\langle \text{Impl } A B \in S \rangle$   
**then have**  $\langle \text{Impl } A B \in x \rangle$   
**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{\text{Neg } A\} \in C \vee x \cup \{B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *simp*  
**then show**  $\langle S \cup \{\text{Neg } A\} \in \text{close } C \vee S \cup \{B\} \in \text{close } C \rangle$   
**using**  $\langle S \subseteq x \rangle$  *subset-in-close* **by** *blast* }

**{ fix**  $A B$   
**assume**  $\langle \text{Neg } (\text{Impl } A B) \in S \rangle$   
**then have**  $\langle \text{Neg } (\text{Impl } A B) \in x \rangle$   
**using**  $\langle S \subseteq x \rangle$  **by** *blast*  
**then have**  $\langle x \cup \{A, \text{Neg } B\} \in C \rangle$   
**using**  $\langle x \in C \rangle$  *conc* **unfolding** *consistency-def* **by** *blast*  
**then show**  $\langle S \cup \{A, \text{Neg } B\} \in \text{close } C \rangle$

```

    using  $\langle S \subseteq x \rangle$  subset-in-close by blast }

{ fix  $P$  and  $t :: \langle 'a \text{ term} \rangle$ 
  assume  $\langle \text{closed } 0 \ t \rangle$  and  $\langle \text{Forall } P \in S \rangle$ 
  then have  $\langle \text{Forall } P \in x \rangle$ 
    using  $\langle S \subseteq x \rangle$  by blast
  then have  $\langle x \cup \{P[t/0]\} \in C \rangle$ 
    using  $\langle \text{closed } 0 \ t \rangle$   $\langle x \in C \rangle$  conc unfolding consistency-def by blast
  then show  $\langle S \cup \{P[t/0]\} \in \text{close } C \rangle$ 
    using  $\langle S \subseteq x \rangle$  subset-in-close by blast }

{ fix  $P$  and  $t :: \langle 'a \text{ term} \rangle$ 
  assume  $\langle \text{closed } 0 \ t \rangle$  and  $\langle \text{Neg } (\text{Exists } P) \in S \rangle$ 
  then have  $\langle \text{Neg } (\text{Exists } P) \in x \rangle$ 
    using  $\langle S \subseteq x \rangle$  by blast
  then have  $\langle x \cup \{\text{Neg } (P[t/0])\} \in C \rangle$ 
    using  $\langle \text{closed } 0 \ t \rangle$   $\langle x \in C \rangle$  conc unfolding consistency-def by blast
  then show  $\langle S \cup \{\text{Neg } (P[t/0])\} \in \text{close } C \rangle$ 
    using  $\langle S \subseteq x \rangle$  subset-in-close by blast }

{ fix  $P$ 
  assume  $\langle \text{Exists } P \in S \rangle$ 
  then have  $\langle \text{Exists } P \in x \rangle$ 
    using  $\langle S \subseteq x \rangle$  by blast
  then have  $\langle \exists c. x \cup \{P[\text{App } c \ []/0]\} \in C \rangle$ 
    using  $\langle x \in C \rangle$  conc unfolding consistency-def by blast
  then show  $\langle \exists c. S \cup \{P[\text{App } c \ []/0]\} \in \text{close } C \rangle$ 
    using  $\langle S \subseteq x \rangle$  subset-in-close by blast }

{ fix  $P$ 
  assume  $\langle \text{Neg } (\text{Forall } P) \in S \rangle$ 
  then have  $\langle \text{Neg } (\text{Forall } P) \in x \rangle$ 
    using  $\langle S \subseteq x \rangle$  by blast
  then have  $\langle \exists c. x \cup \{\text{Neg } (P[\text{App } c \ []/0])\} \in C \rangle$ 
    using  $\langle x \in C \rangle$  conc unfolding consistency-def by simp
  then show  $\langle \exists c. S \cup \{\text{Neg } (P[\text{App } c \ []/0])\} \in \text{close } C \rangle$ 
    using  $\langle S \subseteq x \rangle$  subset-in-close by blast }
qed

```

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

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

**shows**  $\langle \text{subset-closed } (mk\text{-alt-consistency } C) \rangle$   
**unfolding**  $\text{subset-closed-def } mk\text{-alt-consistency-def}$   
**proof**  $(intro\ ballI\ allI\ impI)$   
**fix**  $S\ S' :: \langle ('a, 'b)\ \text{form set} \rangle$   
**assume**  $\langle S \subseteq S' \rangle$  **and**  $\langle S' \in \{S. \exists f. psubst\ f\ 'S \in C\} \rangle$   
**then obtain**  $f$  **where**  $*$ :  $\langle psubst\ f\ 'S' \in C \rangle$   
**by**  $blast$   
**moreover have**  $\langle psubst\ f\ 'S \subseteq psubst\ f\ 'S' \rangle$   
**using**  $\langle S \subseteq S' \rangle$  **by**  $blast$   
**moreover have**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$   
**using**  $\langle \text{subset-closed } C \rangle$  **unfolding**  $\text{subset-closed-def}$  **by**  $blast$   
**ultimately have**  $\langle psubst\ f\ 'S \in C \rangle$   
**by**  $blast$   
**then show**  $\langle S \in \{S. \exists f. psubst\ f\ 'S \in C\} \rangle$   
**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**  $finite\text{-char} :: \langle 'a\ \text{set set} \Rightarrow bool \rangle$  **where**  
 $\langle finite\text{-char } C = (\forall S. S \in C = (\forall S'. finite\ S' \longrightarrow S' \subseteq S \longrightarrow S' \in C)) \rangle$

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

**theorem**  $finite\text{-alt-consistency}$ :  
**assumes**  $altconc$ :  $\langle alt\text{-consistency } C \rangle$   
**and**  $\langle \text{subset-closed } C \rangle$   
**shows**  $\langle alt\text{-consistency } (mk\text{-finite-char } C) \rangle$   
**unfolding**  $alt\text{-consistency-def}$   
**proof**  $(intro\ allI\ impI\ conjI)$   
**fix**  $S$   
**assume**  $\langle S \in mk\text{-finite-char } C \rangle$   
**then have**  $finC$ :  $\langle \forall S' \subseteq S. finite\ S' \longrightarrow S' \in C \rangle$   
**unfolding**  $mk\text{-finite-char-def}$  **by**  $blast$

**have**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$   
**using**  $\langle \text{subset-closed } C \rangle$  **unfolding**  $\text{subset-closed-def}$  **by**  $blast$   
**then have**  $sc$ :  $\langle \forall S' x. S' \cup x \in C \longrightarrow (\forall S \subseteq S' \cup x. S \in C) \rangle$   
**by**  $blast$

**{ fix**  $p\ ts$   
**show**  $\langle \neg (Pred\ p\ ts \in S \wedge Neg\ (Pred\ p\ ts) \in S) \rangle$   
**proof**

```

assume  $\langle \text{Pred } p \text{ ts} \in S \wedge \text{Neg } (\text{Pred } p \text{ ts}) \in S \rangle$ 
then have  $\langle \{\text{Pred } p \text{ ts}, \text{Neg } (\text{Pred } p \text{ ts})\} \in C \rangle$ 
  using finc by simp
then show False
  using altconc unfolding alt-consistency-def by fast
qed }

```

```

show  $\langle FF \notin S \rangle$ 
proof
  assume  $\langle FF \in S \rangle$ 
  then have  $\langle \{FF\} \in C \rangle$ 
    using finc by simp
  then show False
    using altconc unfolding alt-consistency-def by fast
qed

```

```

show  $\langle \text{Neg } TT \notin S \rangle$ 
proof
  assume  $\langle \text{Neg } TT \in S \rangle$ 
  then have  $\langle \{\text{Neg } TT\} \in C \rangle$ 
    using finc by simp
  then show False
    using altconc unfolding alt-consistency-def by fast
qed

```

```

{ fix Z
  assume *:  $\langle \text{Neg } (\text{Neg } Z) \in S \rangle$ 
  show  $\langle S \cup \{Z\} \in \text{mk-finite-char } C \rangle$ 
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix S'
    let  $?S' = \langle S' - \{Z\} \cup \{\text{Neg } (\text{Neg } Z)\} \rangle$ 

    assume  $\langle S' \subseteq S \cup \{Z\} \rangle$  and  $\langle \text{finite } S' \rangle$ 
    then have  $\langle ?S' \subseteq S \rangle$ 
      using * by blast
    moreover have  $\langle \text{finite } ?S' \rangle$ 
      using  $\langle \text{finite } S' \rangle$  by blast
    ultimately have  $\langle ?S' \in C \rangle$ 
      using finc by blast
    then have  $\langle ?S' \cup \{Z\} \in C \rangle$ 
      using altconc unfolding alt-consistency-def by simp
    then show  $\langle S' \in C \rangle$ 
      using sc by blast
  qed }

```

```

{ fix A B
  assume *:  $\langle \text{And } A \ B \in S \rangle$ 
  show  $\langle S \cup \{A, B\} \in \text{mk-finite-char } C \rangle$ 

```

```

unfolding mk-finite-char-def
proof (intro allI impI CollectI)
fix  $S'$ 
let  $?S' = \langle S' - \{A, B\} \cup \{And\ A\ B\} \rangle$ 

assume  $\langle S' \subseteq S \cup \{A, B\} \rangle$  and  $\langle finite\ S' \rangle$ 
then have  $\langle ?S' \subseteq S \rangle$ 
  using * by blast
moreover have  $\langle finite\ ?S' \rangle$ 
  using  $\langle finite\ S' \rangle$  by blast
ultimately have  $\langle ?S' \in C \rangle$ 
  using fine by blast
then have  $\langle ?S' \cup \{A, B\} \in C \rangle$ 
  using altconc unfolding alt-consistency-def by simp
then show  $\langle S' \in C \rangle$ 
  using sc by blast
qed }

{ fix  $A\ B$ 
assume *:  $\langle Neg\ (Or\ A\ B) \in S \rangle$ 
show  $\langle S \cup \{Neg\ A,\ Neg\ B\} \in mk-finite-char\ C \rangle$ 
  unfolding mk-finite-char-def
proof (intro allI impI CollectI)
fix  $S'$ 
let  $?S' = \langle S' - \{Neg\ A,\ Neg\ B\} \cup \{Neg\ (Or\ A\ B)\} \rangle$ 

assume  $\langle S' \subseteq S \cup \{Neg\ A,\ Neg\ B\} \rangle$  and  $\langle finite\ S' \rangle$ 
then have  $\langle ?S' \subseteq S \rangle$ 
  using * by blast
moreover have  $\langle finite\ ?S' \rangle$ 
  using  $\langle finite\ S' \rangle$  by blast
ultimately have  $\langle ?S' \in C \rangle$ 
  using fine by blast
then have  $\langle ?S' \cup \{Neg\ A,\ Neg\ B\} \in C \rangle$ 
  using altconc unfolding alt-consistency-def by simp
then show  $\langle S' \in C \rangle$ 
  using sc by blast
qed }

{ fix  $A\ B$ 
assume *:  $\langle Neg\ (Impl\ A\ B) \in S \rangle$ 
show  $\langle S \cup \{A,\ Neg\ B\} \in mk-finite-char\ C \rangle$ 
  unfolding mk-finite-char-def
proof (intro allI impI CollectI)
fix  $S'$ 
let  $?S' = \langle S' - \{A,\ Neg\ B\} \cup \{Neg\ (Impl\ A\ B)\} \rangle$ 

assume  $\langle S' \subseteq S \cup \{A,\ Neg\ B\} \rangle$  and  $\langle finite\ S' \rangle$ 
then have  $\langle ?S' \subseteq S \rangle$ 

```



**using** \* **by** *blast*  
**moreover have**  $\langle \text{finite } ?S' \rangle$   
**using**  $\langle \text{finite } S' \rangle$  **by** *blast*  
**ultimately have**  $\langle ?S' \in C \rangle$   
**using** *finc* **by** *blast*  
**then have**  $\langle ?S' \cup \{A, \text{Neg } B\} \in C \rangle$   
**using** *altconc unfolding alt-consistency-def* **by** *simp*  
**then show**  $\langle S' \in C \rangle$   
**using** *sc* **by** *blast*  
**qed** }

**{ fix**  $A B$   
**assume** \*:  $\langle \text{Or } A B \in S \rangle$   
**show**  $\langle S \cup \{A\} \in \text{mk-finite-char } C \vee S \cup \{B\} \in \text{mk-finite-char } C \rangle$   
**proof** (*rule ccontr*)  
**assume**  $\langle \neg ?thesis \rangle$   
**then obtain**  $Sa$  **and**  $Sb$   
**where**  $\langle Sa \subseteq S \cup \{A\} \rangle$  **and**  $\langle \text{finite } Sa \rangle$  **and**  $\langle Sa \notin C \rangle$   
**and**  $\langle Sb \subseteq S \cup \{B\} \rangle$  **and**  $\langle \text{finite } Sb \rangle$  **and**  $\langle Sb \notin C \rangle$   
**unfolding** *mk-finite-char-def* **by** *blast*

**let**  $?S' = \langle (Sa - \{A\}) \cup (Sb - \{B\}) \cup \{\text{Or } A B\} \rangle$

**have**  $\langle ?S' \subseteq S \rangle$   
**using**  $\langle Sa \subseteq S \cup \{A\} \rangle$   $\langle Sb \subseteq S \cup \{B\} \rangle$  \* **by** *blast*  
**moreover have**  $\langle \text{finite } ?S' \rangle$   
**using**  $\langle \text{finite } Sa \rangle$   $\langle \text{finite } Sb \rangle$  **by** *blast*  
**ultimately have**  $\langle ?S' \in C \rangle$   
**using** *finc* **by** *blast*  
**then have**  $\langle ?S' \cup \{A\} \in C \vee ?S' \cup \{B\} \in C \rangle$   
**using** *altconc unfolding alt-consistency-def* **by** *simp*  
**then have**  $\langle Sa \in C \vee Sb \in C \rangle$   
**using** *sc* **by** *blast*  
**then show** *False*  
**using**  $\langle Sa \notin C \rangle$   $\langle Sb \notin C \rangle$  **by** *blast*  
**qed** }

**{ fix**  $A B$   
**assume** \*:  $\langle \text{Neg } (\text{And } A B) \in S \rangle$   
**show**  $\langle S \cup \{\text{Neg } A\} \in \text{mk-finite-char } C \vee S \cup \{\text{Neg } B\} \in \text{mk-finite-char } C \rangle$   
**proof** (*rule ccontr*)  
**assume**  $\langle \neg ?thesis \rangle$   
**then obtain**  $Sa$  **and**  $Sb$   
**where**  $\langle Sa \subseteq S \cup \{\text{Neg } A\} \rangle$  **and**  $\langle \text{finite } Sa \rangle$  **and**  $\langle Sa \notin C \rangle$   
**and**  $\langle Sb \subseteq S \cup \{\text{Neg } B\} \rangle$  **and**  $\langle \text{finite } Sb \rangle$  **and**  $\langle Sb \notin C \rangle$   
**unfolding** *mk-finite-char-def* **by** *blast*

**let**  $?S' = \langle (Sa - \{\text{Neg } A\}) \cup (Sb - \{\text{Neg } B\}) \cup \{\text{Neg } (\text{And } A B)\} \rangle$

```

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 ⟨ $?S' \subseteq S$ ⟩
  using * by blast
moreover have ⟨finite  $?S'$ ⟩
  using ⟨finite  $S'$ ⟩ by blast
ultimately have ⟨ $?S' \in C$ ⟩
  using finc by blast
then have ⟨ $?S' \cup \{P[t/0]\} \in C$ ⟩
  using altconc ⟨closedt 0 t⟩ unfolding alt-consistency-def by simp
then show ⟨ $S' \in C$ ⟩
  using sc by blast
qed }

```

```

{ fix  $P$  and  $t :: \langle 'a \text{ term} \rangle$ 
  assume *: ⟨ $Neg (Exists P) \in S$ ⟩ and ⟨closedt 0 t⟩
  show ⟨ $S \cup \{Neg (P[t/0])\} \in mk\text{-finite-char } C$ ⟩
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix  $S'$ 
    let  $?S' = \langle S' - \{Neg (P[t/0])\} \cup \{Neg (Exists P)\} \rangle$ 

    assume ⟨ $S' \subseteq S \cup \{Neg (P[t/0])\}$ ⟩ and ⟨finite  $S'$ ⟩
    then have ⟨ $?S' \subseteq S$ ⟩
      using * by blast
    moreover have ⟨finite  $?S'$ ⟩
      using ⟨finite  $S'$ ⟩ by blast
    ultimately have ⟨ $?S' \in C$ ⟩
      using finc by blast
    then have ⟨ $?S' \cup \{Neg (P[t/0])\} \in C$ ⟩
      using altconc ⟨closedt 0 t⟩ unfolding alt-consistency-def by simp
    then show ⟨ $S' \in C$ ⟩
      using sc by blast
  qed }

```

```

{ fix  $P x$ 
  assume *: ⟨ $Exists P \in S$ ⟩ and ⟨ $\forall a \in S. x \notin \text{params } a$ ⟩
  show ⟨ $S \cup \{P[App x []/0]\} \in mk\text{-finite-char } C$ ⟩
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix  $S'$ 
    let  $?S' = \langle S' - \{P[App x []/0]\} \cup \{Exists P\} \rangle$ 

    assume ⟨ $S' \subseteq S \cup \{P[App x []/0]\}$ ⟩ and ⟨finite  $S'$ ⟩
    then have ⟨ $?S' \subseteq S$ ⟩
      using * by blast
    moreover have ⟨finite  $?S'$ ⟩
      using ⟨finite  $S'$ ⟩ by blast
    ultimately have ⟨ $?S' \in C$ ⟩
      using finc by blast
    moreover have ⟨ $\forall a \in ?S'. x \notin \text{params } a$ ⟩

```

```

    using  $\langle \forall a \in S. x \notin \text{params } a \rangle \langle ?S' \subseteq S \rangle$  by blast
  ultimately have  $\langle ?S' \cup \{P[\text{App } x \ \_ / 0]\} \in C \rangle$ 
    using altconc  $\langle \forall a \in S. x \notin \text{params } a \rangle$  unfolding alt-consistency-def by
blast
  then show  $\langle S' \in C \rangle$ 
    using sc by blast
  qed }

{ fix  $P x$ 
  assume *:  $\langle \text{Neg } (\text{Forall } P) \in S \rangle$  and  $\langle \forall a \in S. x \notin \text{params } a \rangle$ 
  show  $\langle S \cup \{ \text{Neg } (P[\text{App } x \ \_ / 0]) \} \in \text{mk-finite-char } C \rangle$ 
    unfolding mk-finite-char-def
  proof (intro allI impI CollectI)
    fix  $S'$ 
    let  $?S' = \langle S' - \{ \text{Neg } (P[\text{App } x \ \_ / 0]) \} \cup \{ \text{Neg } (\text{Forall } P) \} \rangle$ 

    assume  $\langle S' \subseteq S \cup \{ \text{Neg } (P[\text{App } x \ \_ / 0]) \} \rangle$  and  $\langle \text{finite } S' \rangle$ 
    then have  $\langle ?S' \subseteq S \rangle$ 
      using * by blast
    moreover have  $\langle \text{finite } ?S' \rangle$ 
      using  $\langle \text{finite } S' \rangle$  by blast
    ultimately have  $\langle ?S' \in C \rangle$ 
      using finc by blast
    moreover have  $\langle \forall a \in ?S'. x \notin \text{params } a \rangle$ 
      using  $\langle \forall a \in S. x \notin \text{params } a \rangle \langle ?S' \subseteq S \rangle$  by blast
    ultimately have  $\langle ?S' \cup \{ \text{Neg } (P[\text{App } x \ \_ / 0]) \} \in C \rangle$ 
      using altconc  $\langle \forall a \in S. x \notin \text{params } a \rangle$  unfolding alt-consistency-def by
simp
    then show  $\langle S' \in C \rangle$ 
      using sc by blast
    qed }
  qed

theorem finite-char:  $\langle \text{finite-char } (\text{mk-finite-char } C) \rangle$ 
  unfolding finite-char-def mk-finite-char-def by blast

theorem finite-char-closed:  $\langle \text{finite-char } C \implies \text{subset-closed } C \rangle$ 
  unfolding finite-char-def subset-closed-def
  proof (intro ballI allI impI)
    fix  $S S'$ 
    assume *:  $\langle \forall S. (S \in C) = (\forall S'. \text{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in C) \rangle$ 
      and  $\langle S' \in C \rangle$  and  $\langle S \subseteq S' \rangle$ 
    then have  $\langle \forall S'. \text{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in C \rangle$  by blast
    then show  $\langle S \in C \rangle$  using * by blast
  qed

theorem finite-char-subset:  $\langle \text{subset-closed } C \implies C \subseteq \text{mk-finite-char } C \rangle$ 
  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**  $\langle m \leq \max n m \rangle$  **and**  $\langle n \leq \max n m \rangle$   
**by** *simp-all*  
**have**  $\langle x \in f (\max n m) \rangle$   
**using** *is-chainD' ch*  $\langle x \in f m \rangle$   $\langle m \leq \max n m \rangle$  **by** *fast*  
**moreover have**  $\langle F \subseteq f (\max n m) \rangle$   
**using** *is-chainD' ch*  $\langle F \subseteq f n \rangle$   $\langle n \leq \max n m \rangle$  **by** *fast*  
**moreover have**  $\langle x \in f (\max n m) \wedge F \subseteq f (\max n m) \rangle$   
**using** *calculation* **by** *blast*  
**ultimately show** *?case* **by** *blast*  
**qed**

**lemma** *chain-union-closed'*:  
**assumes**  $\langle \text{is-chain } f \rangle$  **and**  $\langle (\forall n. f n \in C) \rangle$  **and**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$   
**and**  $\langle \text{finite } S' \rangle$  **and**  $\langle S' \subseteq (\bigcup n. f n) \rangle$   
**shows**  $\langle S' \in C \rangle$   
**proof** –  
**note**  $\langle \text{finite } S' \rangle$  **and**  $\langle S' \subseteq (\bigcup n. f n) \rangle$   
**then obtain**  $n$  **where**  $\langle S' \subseteq f n \rangle$   
**using** *chain-index*  $\langle \text{is-chain } f \rangle$  **by** *blast*  
**moreover have**  $\langle f n \in C \rangle$   
**using**  $\langle \forall n. f n \in C \rangle$  **by** *blast*  
**ultimately show**  $\langle S' \in C \rangle$   
**using**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$  **by** *blast*  
**qed**

**theorem** *chain-union-closed*:  
**assumes**  $\langle \text{finite-char } C \rangle$  **and**  $\langle \text{is-chain } f \rangle$  **and**  $\langle \forall n. f n \in C \rangle$   
**shows**  $\langle (\bigcup n. f n) \in C \rangle$   
**proof** –  
**have**  $\langle \text{subset-closed } C \rangle$   
**using** *finite-char-closed*  $\langle \text{finite-char } C \rangle$  **by** *blast*  
**then have**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$   
**using** *subset-closed-def* **by** *blast*  
**then have**  $\langle \forall S'. \text{finite } S' \longrightarrow S' \subseteq (\bigcup n. f n) \longrightarrow S' \in C \rangle$   
**using** *chain-union-closed' assms* **by** *blast*  
**moreover have**  $\langle ((\bigcup n. f n) \in C) = (\forall S'. \text{finite } S' \longrightarrow S' \subseteq (\bigcup n. f n) \longrightarrow S' \in C) \rangle$   
**using**  $\langle \text{finite-char } C \rangle$  **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* ::  $\langle ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$  **where**  
 $\langle \text{dest-Neg } (\text{Neg } p) = p \rangle$

**primrec** (*nonexhaustive*) *dest-Forall* ::  $\langle ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$  **where**  
 $\langle \text{dest-Forall } (\text{Forall } p) = p \rangle$

**primrec** (*nonexhaustive*) *dest-Exists* ::  $\langle ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$  **where**  
 $\langle \text{dest-Exists } (\text{Exists } p) = p \rangle$

**primrec** *extend* ::  $\langle (\text{nat}, 'b) \text{ form set} \Rightarrow (\text{nat}, 'b) \text{ form set set} \Rightarrow$   
 $(\text{nat} \Rightarrow (\text{nat}, 'b) \text{ form}) \Rightarrow \text{nat} \Rightarrow (\text{nat}, 'b) \text{ form set} \rangle$  **where**  
 $\langle \text{extend } S \ C \ f \ 0 = S \rangle$   
 $| \langle \text{extend } S \ C \ f \ (\text{Suc } n) = (\text{if } \text{extend } S \ C \ f \ n \cup \{f \ n\} \in C$   
*then*  
 $(\text{if } (\exists p. f \ n = \text{Exists } p)$   
 $\text{then } \text{extend } S \ C \ f \ n \cup \{f \ n\} \cup \{\text{subst } (\text{dest-Exists } (f \ n))$   
 $(\text{App } (\text{SOME } k. k \notin (\bigcup p \in \text{extend } S \ C \ f \ n \cup \{f \ n\}. \text{params } p)) \ [] ) \ 0\}$   
 $\text{else if } (\exists p. f \ n = \text{Neg } (\text{Forall } p))$   
 $\text{then } \text{extend } S \ C \ f \ n \cup \{f \ n\} \cup \{\text{Neg } (\text{subst } (\text{dest-Forall } (\text{dest-Neg } (f \ n)))$   
 $(\text{App } (\text{SOME } k. k \notin (\bigcup p \in \text{extend } S \ C \ f \ n \cup \{f \ n\}. \text{params } p)) \ [] ) \ 0\}$   
 $\text{else } \text{extend } S \ C \ f \ n \cup \{f \ n\}$   
 $\text{else } \text{extend } S \ C \ f \ n \rangle$

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

**theorem** *is-chain-extend*:  $\langle \text{is-chain } (\text{extend } S \ C \ f) \rangle$   
**by** (*simp add: is-chain-def*) *blast*

**theorem** *finite-paramst* [*simp*]:  $\langle \text{finite } (\text{paramst } (t :: 'a \text{ term})) \rangle$   
 $\langle \text{finite } (\text{paramsts } (ts :: 'a \text{ term list})) \rangle$   
**by** (*induct t and ts rule: paramst.induct paramsts.induct*) (*simp-all split: sum.split*)

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

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

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

**have**  $\langle \text{infinite } (\neg (\bigcup x \in ?S'. \text{params } x)) \rangle$   
**using** *assms* **by** *simp*  
**then obtain**  $x$  **where**  $\langle x \in \neg (\bigcup x \in ?S'. \text{params } x) \rangle$   
**using** *infinite-imp-nonempty* **by** *blast*  
**then have**  $\langle \forall a \in ?S'. x \notin \text{params } a \rangle$   
**by** *blast*  
**then show** *?thesis*

by *blast*  
qed

**lemma** *extend-in-C-Exists*:

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

**proof** –

obtain  $p$  where  $*$ :  $\langle f \ n = \text{Exists } p \rangle$   
 using  $\langle \exists p. f \ n = \text{Exists } p \rangle$  by *blast*  
 have  $\langle \exists x. x \notin (\bigcup p \in ?S'. \text{params } p) \rangle$   
 using  $\langle \text{infinite } (- (\bigcup p \in S. \text{params } p)) \rangle$  *infinite-params-available*  
 by *blast*  
 moreover have  $\langle \text{Exists } p \in ?S' \rangle$   
 using  $*$  by *simp*  
 then have  $\langle \forall x. x \notin (\bigcup p \in ?S'. \text{params } p) \longrightarrow ?S' \cup \{p[\text{App } x \ []/0]\} \in C \rangle$   
 using  $\langle ?S' \in C \rangle$   $\langle \text{alt-consistency } C \rangle$   
 unfolding *alt-consistency-def* by *simp*  
 ultimately have  $\langle (?S' \cup \{p[\text{App } (SOME \ k. k \notin (\bigcup p \in ?S'. \text{params } p)) \ []/0]\}) \in C \rangle$   
 by (*metis* (*mono-tags*, *lifting*) *someI2*)  
 then show *?thesis*  
 using *assms*  $*$  by *simp*

qed

**lemma** *extend-in-C-Neg-Forall*:

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

**proof** –

obtain  $p$  where  $*$ :  $\langle f \ n = \text{Neg } (\text{Forall } p) \rangle$   
 using  $\langle \exists p. f \ n = \text{Neg } (\text{Forall } p) \rangle$  by *blast*  
 have  $\langle \exists x. x \notin (\bigcup p \in ?S'. \text{params } p) \rangle$   
 using  $\langle \text{infinite } (- (\bigcup p \in S. \text{params } p)) \rangle$  *infinite-params-available*  
 by *blast*  
 moreover have  $\langle \text{Neg } (\text{Forall } p) \in ?S' \rangle$   
 using  $*$  by *simp*  
 then have  $\langle \forall x. x \notin (\bigcup p \in ?S'. \text{params } p) \longrightarrow ?S' \cup \{\text{Neg } (p[\text{App } x \ []/0])\} \in C \rangle$   
 using  $\langle ?S' \in C \rangle$   $\langle \text{alt-consistency } C \rangle$   
 unfolding *alt-consistency-def* by *simp*  
 ultimately have  $\langle (?S' \cup \{\text{Neg } (p[\text{App } (SOME \ k. k \notin (\bigcup p \in ?S'. \text{params } p)) \ []/0])\}) \in C \rangle$   
 by (*metis* (*mono-tags*, *lifting*) *someI2*)



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

**lemma** *extend-in-C-no-delta*:  
**assumes**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \in C \rangle$   
**and**  $\langle \forall p. f \ n \neq \text{Exists } p \rangle$   
**and**  $\langle \forall p. f \ n \neq \text{Neg } (\text{Forall } p) \rangle$   
**shows**  $\langle \text{extend } S \ C \ f \ (\text{Suc } n) \in C \rangle$   
**using** *assms by simp*

**lemma** *extend-in-C-stop*:  
**assumes**  $\langle \text{extend } S \ C \ f \ n \in C \rangle$   
**and**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \notin C \rangle$   
**shows**  $\langle \text{extend } S \ C \ f \ (\text{Suc } n) \in C \rangle$   
**using** *assms by simp*

**theorem** *extend-in-C*:  $\langle \text{alt-consistency } C \implies$   
 $S \in C \implies \text{infinite } (- (\bigcup p \in S. \text{params } p)) \implies \text{extend } S \ C \ f \ n \in C \rangle$   
**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  $\text{Extend } S \ C \ f$  is again consistent.

**theorem** *Extend-in-C*:  $\langle \text{alt-consistency } C \implies \text{finite-char } C \implies$   
 $S \in C \implies \text{infinite } (- (\bigcup p \in S. \text{params } p)) \implies \text{Extend } S \ C \ f \in C \rangle$   
**unfolding** *Extend-def*  
**using** *chain-union-closed is-chain-extend extend-in-C*  
**by** *blast*

**theorem** *Extend-subset*:  $\langle S \subseteq \text{Extend } S \ C \ f \rangle$   
**proof**  
**fix**  $x$   
**assume**  $\langle x \in S \rangle$   
**then have**  $\langle x \in \text{extend } S \ C \ f \ 0 \rangle$  **by** *simp*  
**then have**  $\langle \exists n. x \in \text{extend } S \ C \ f \ n \rangle$  **by** *blast*  
**then show**  $\langle x \in \text{Extend } S \ C \ f \rangle$  **by** (*simp add: Extend-def*)  
**qed**

The *Extend* function yields a maximal set:

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

**theorem** *extend-maximal*:

**assumes**  $\langle \forall y. \exists n. y = f \ n \rangle$   
**and**  $\langle \text{finite-char } C \rangle$   
**shows**  $\langle \text{maximal } (\text{Extend } S \ C \ f) \ C \rangle$   
**unfolding** *maximal-def Extend-def*  
**proof** (*intro ballI impI*)  
**fix**  $S'$   
**assume**  $\langle S' \in C \rangle$   
**and**  $\langle (\bigcup x. \text{extend } S \ C \ f \ x) \subseteq S' \rangle$   
**moreover have**  $\langle S' \subseteq (\bigcup x. \text{extend } S \ C \ f \ x) \rangle$   
**proof** (*rule ccontr*)  
**assume**  $\langle \neg S' \subseteq (\bigcup x. \text{extend } S \ C \ f \ x) \rangle$   
**then have**  $\langle \exists z. z \in S' \wedge z \notin (\bigcup x. \text{extend } S \ C \ f \ x) \rangle$   
**by blast**  
**then obtain**  $z$  **where**  $\langle z \in S' \rangle$  **and**  $*$ :  $\langle z \notin (\bigcup x. \text{extend } S \ C \ f \ x) \rangle$   
**by blast**  
**then obtain**  $n$  **where**  $\langle z = f \ n \rangle$   
**using**  $\langle \forall y. \exists n. y = f \ n \rangle$  **by blast**  
  
**from**  $\langle (\bigcup x. \text{extend } S \ C \ f \ x) \subseteq S' \rangle \langle z = f \ n \rangle \langle z \in S' \rangle$   
**have**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \subseteq S' \rangle$  **by blast**  
  
**from**  $\langle \text{finite-char } C \rangle$   
**have**  $\langle \text{subset-closed } C \rangle$  **using** *finite-char-closed* **by blast**  
**then have**  $\langle \forall S' \in C. \forall S \subseteq S'. S \in C \rangle$   
**unfolding** *subset-closed-def* **by simp**  
**then have**  $\langle \forall S \subseteq S'. S \in C \rangle$   
**using**  $\langle S' \in C \rangle$  **by blast**  
**then have**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \in C \rangle$   
**using**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \subseteq S' \rangle$   
**by blast**  
**then have**  $\langle z \in \text{extend } S \ C \ f \ (\text{Suc } n) \rangle$   
**using**  $\langle z \notin (\bigcup x. \text{extend } S \ C \ f \ x) \rangle \langle z = f \ n \rangle$   
**by simp**  
**then show** *False* **using**  $*$  **by blast**  
**qed**  
**ultimately show**  $\langle (\bigcup x. \text{extend } S \ C \ f \ x) = S' \rangle$   
**by simp**  
**qed**

## 7.6 Hintikka sets and Herbrand models

A Hintikka set is defined as follows:

**definition** *hintikka* ::  $\langle ('a, 'b) \text{ form set} \Rightarrow \text{bool} \rangle$  **where**  
 $\langle \text{hintikka } H =$   
 $((\forall p \ ts. \neg (\text{Pred } p \ ts \in H \wedge \text{Neg } (\text{Pred } p \ ts) \in H)) \wedge$

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

In Herbrand models, each *closed* term is interpreted by itself. We introduce a new datatype *hterm* (“Herbrand terms”), which is similar to the datatype *term* introduced in §3, but without variables. We also define functions for converting between closed terms and Herbrand terms.

**datatype** *'a hterm* = *HApp 'a <'a hterm list>*

**primrec**

*term-of-hterm* :: *<'a hterm ⇒ 'a term>* **and**  
*terms-of-hterms* :: *<'a hterm list ⇒ 'a term list>* **where**  
*<term-of-hterm (HApp a hts) = App a (terms-of-hterms hts)>*  
| *<terms-of-hterms [] = []>*  
| *<terms-of-hterms (ht # hts) = term-of-hterm ht # terms-of-hterms hts>*

**theorem** *herbrand-evalt* [*simp*]:

*<closedt 0 t ⇒ term-of-hterm (evalt e HApp t) = t>*  
*<closedts 0 ts ⇒ terms-of-hterms (evalts e HApp ts) = ts>*  
**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 ⟨ $p \in H \longrightarrow \text{closed } 0 \ p \longrightarrow$ 
   $\text{eval } e \ \text{HApp } (\lambda a \ ts. \text{Pred } a \ (\text{terms-of-hterms } ts) \in H) \ p \rangle \wedge$ 
  ⟨ $\text{Neg } p \in H \longrightarrow \text{closed } 0 \ p \longrightarrow$ 
   $\text{eval } e \ \text{HApp } (\lambda a \ ts. \text{Pred } a \ (\text{terms-of-hterms } ts) \in H) \ (\text{Neg } p) \rangle$ 
proof (induct p rule: wf-induct [where r=⟨measure size-form⟩])
  show ⟨wf (measure size-form)⟩
  by blast
next
  let ?eval = ⟨ $\text{eval } e \ \text{HApp } (\lambda a \ ts. \text{Pred } a \ (\text{terms-of-hterms } ts) \in H) \rangle$ 

  fix x
  assume wf: ⟨ $\forall y. (y, x) \in \text{measure size-form} \longrightarrow$ 
     $(y \in H \longrightarrow \text{closed } 0 \ y \longrightarrow ?\text{eval } y) \wedge$ 
     $(\text{Neg } y \in H \longrightarrow \text{closed } 0 \ y \longrightarrow ?\text{eval } (\text{Neg } y)) \rangle$ 

  show ⟨ $(x \in H \longrightarrow \text{closed } 0 \ x \longrightarrow ?\text{eval } x) \wedge (\text{Neg } x \in H \longrightarrow \text{closed } 0 \ x \longrightarrow$ 
     $?\text{eval } (\text{Neg } x)) \rangle$ 
  proof (cases x)
    case FF
      show ?thesis
      proof (intro conjI impI)
        assume ⟨ $x \in H$ ⟩
        then show ⟨ $?\text{eval } x$ ⟩
        using FF hin by (simp add: hintikka-def)
      next
        assume ⟨ $\text{Neg } x \in H$ ⟩
        then show ⟨ $?\text{eval } (\text{Neg } x)$ ⟩ using FF by simp
      qed
    next
      case TT
        show ?thesis
        proof (intro conjI impI)
          assume ⟨ $x \in H$ ⟩
          then show ⟨ $?\text{eval } x$ ⟩
          using TT by simp
        next
          assume ⟨ $\text{Neg } x \in H$ ⟩
          then show ⟨ $?\text{eval } (\text{Neg } x)$ ⟩
          using TT hin by (simp add: hintikka-def)
        qed
    next
      case (Pred p ts)
        show ?thesis
        proof (intro conjI impI)
          assume ⟨ $x \in H$ ⟩ and ⟨ $\text{closed } 0 \ x$ ⟩
          then show ⟨ $?\text{eval } x$ ⟩ using Pred by simp
        next
          assume ⟨ $\text{Neg } x \in H$ ⟩ and ⟨ $\text{closed } 0 \ x$ ⟩

```

```

    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  $\langle x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
  then have  $\langle Or \ A \ B \in H \rangle$  and  $\langle \text{closed } 0 \ (Or \ A \ B) \rangle$ 
    using Or by simp-all
  then have  $\langle A \in H \vee B \in H \rangle$ 
    using hin unfolding hintikka-def by blast
  then show  $\langle ?eval \ x \rangle$ 
    using Or wf  $\langle \text{closed } 0 \ (Or \ A \ B) \rangle$  by fastforce
next
  assume  $\langle Neg \ x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
  then have  $\langle Neg \ (Or \ A \ B) \in H \rangle$  and  $\langle \text{closed } 0 \ (Or \ A \ B) \rangle$ 
    using Or by simp-all
  then have  $\langle Neg \ A \in H \wedge Neg \ B \in H \rangle$ 
    using hin unfolding hintikka-def by blast
  then show  $\langle ?eval \ (Neg \ x) \rangle$ 
    using Or wf  $\langle \text{closed } 0 \ (Or \ A \ B) \rangle$  by simp
qed
next
case (Impl A B)
then show ?thesis
proof (intro conjI impI)
  assume  $\langle x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
  then have  $\langle Impl \ A \ B \in H \rangle$  and  $\langle \text{closed } 0 \ (Impl \ A \ B) \rangle$ 
    using Impl by simp-all
  then have  $\langle Neg \ A \in H \vee B \in H \rangle$ 
    using hin unfolding hintikka-def by blast
  then show  $\langle ?eval \ x \rangle$ 
    using Impl wf  $\langle \text{closed } 0 \ (Impl \ A \ B) \rangle$  by fastforce
next
  assume  $\langle Neg \ x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
  then have  $\langle Neg \ (Impl \ A \ B) \in H \rangle$  and  $\langle \text{closed } 0 \ (Impl \ A \ B) \rangle$ 
    using Impl by simp-all
  then have  $\langle A \in H \wedge Neg \ B \in H \rangle$ 
    using hin unfolding hintikka-def by blast
  then show  $\langle ?eval \ (Neg \ x) \rangle$ 
    using Impl wf  $\langle \text{closed } 0 \ (Impl \ A \ B) \rangle$  by simp
qed
next
case (Forall P)
then show ?thesis
proof (intro conjI impI)
  assume  $\langle x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
  have  $\langle \forall z. \ eval \ (e(0:z)) \ HApp \ (\lambda a \ ts. \ Pred \ a \ (terms-of-hterms \ ts) \in H) \ P \rangle$ 
  proof (rule allI)
    fix z
    from  $\langle x \in H \rangle$  and  $\langle \text{closed } 0 \ x \rangle$ 
    have  $\langle Forall \ P \in H \rangle$  and  $\langle \text{closed } 0 \ (Forall \ P) \rangle$ 
      using Forall by simp-all
    then have  $\ast: \langle \forall P \ t. \ closedt \ 0 \ t \longrightarrow Forall \ P \in H \longrightarrow P[t/0] \in H \rangle$ 
  qed

```

```

    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. closed 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**  $\langle \exists p. f\ n = \text{Exists } p \rangle$   
**using**  $\langle \text{Exists } P = f\ n \rangle$  **by** *metis*  
**then have**  $\langle \text{extend } S\ C\ f\ (\text{Suc } n) = (?S' \cup \{(\text{dest-Exists } (f\ n))[?t/0]\}) \rangle$   
**using**  $\langle ?S' \in C \rangle$  **by** *simp*  
**also have**  $\langle \dots = (?S' \cup \{(\text{dest-Exists } (\text{Exists } P))[?t/0]\}) \rangle$   
**using**  $\langle \text{Exists } P = f\ n \rangle$  **by** *simp*  
**also have**  $\langle \dots = (?S' \cup \{P[?t/0]\}) \rangle$   
**by** *simp*  
**finally show** *?thesis*  
**by** *blast*  
**qed**

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

**proof** –  
**have**  $\langle f\ n \neq \text{Exists } P \rangle$   
**using**  $\langle \text{Neg } (\text{Forall } P) = f\ n \rangle$  **by** *auto*  
  
**have**  $\langle \exists p. f\ n = \text{Neg } (\text{Forall } p) \rangle$   
**using**  $\langle \text{Neg } (\text{Forall } P) = f\ n \rangle$  **by** *metis*  
**then have**  $\langle \text{extend } S\ C\ f\ (\text{Suc } n) = (?S' \cup \{ \text{Neg } (\text{dest-Forall } (\text{dest-Neg } (f\ n)))[?t/0] \}) \rangle$   
**using**  $\langle ?S' \in C \rangle$   $\langle f\ n \neq \text{Exists } P \rangle$  **by** *auto*  
**also have**  $\langle \dots = (?S' \cup \{ \text{Neg } (\text{dest-Forall } (\text{dest-Neg } (\text{Neg } (\text{Forall } P)))[?t/0] \}) \rangle$   
**using**  $\langle \text{Neg } (\text{Forall } P) = f\ n \rangle$  **by** *simp*  
**also have**  $\langle \dots = (?S' \cup \{ \text{Neg } (P[?t/0]) \}) \rangle$   
**by** *simp*  
**finally show** *?thesis*  
**by** *blast*  
**qed**

**theorem** *extend-hintikka*:  
**assumes** *fin-ch*:  $\langle \text{finite-char } C \rangle$   
**and** *infin-p*:  $\langle \text{infinite } (\neg (\bigcup p \in S. \text{params } p)) \rangle$   
**and** *surj*:  $\langle \forall y. \exists n. y = f\ n \rangle$   
**and** *altc*:  $\langle \text{alt-consistency } C \rangle$   
**and**  $\langle S \in C \rangle$   
**shows**  $\langle \text{hintikka } (\text{Extend } S\ C\ f) \rangle$  (**is**  $\langle \text{hintikka } ?H \rangle$ )  
**unfolding** *hintikka-def*  
**proof** (*intro allI impI conjI*)  
**have**  $\langle \text{maximal } ?H\ C \rangle$   
**by** (*simp add: extend-maximal fin-ch surj*)

```

have  $\langle ?H \in C \rangle$ 
  using Extend-in-C assms by blast

have  $\langle \forall S' \in C. ?H \subseteq S' \longrightarrow ?H = S' \rangle$ 
  using  $\langle \text{maximal } ?H C \rangle$ 
  unfolding maximal-def by blast

{ fix  $p\ ts$ 
  show  $\langle \neg (Pred\ p\ ts \in ?H \wedge Neg\ (Pred\ p\ ts) \in ?H) \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by fast }

show  $\langle FF \notin ?H \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by blast

show  $\langle Neg\ TT \notin ?H \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by blast

{ fix  $Z$ 
  assume  $\langle Neg\ (Neg\ Z) \in ?H \rangle$ 
  then have  $\langle ?H \cup \{Z\} \in C \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by fast
  then show  $\langle Z \in ?H \rangle$ 
  using  $\langle \text{maximal } ?H C \rangle$  unfolding maximal-def by fast }

{ fix  $A\ B$ 
  assume  $\langle And\ A\ B \in ?H \rangle$ 
  then have  $\langle ?H \cup \{A, B\} \in C \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by fast
  then show  $\langle A \in ?H \rangle$  and  $\langle B \in ?H \rangle$ 
  using  $\langle \text{maximal } ?H C \rangle$  unfolding maximal-def by fast+ }

{ fix  $A\ B$ 
  assume  $\langle Neg\ (Or\ A\ B) \in ?H \rangle$ 
  then have  $\langle ?H \cup \{Neg\ A, Neg\ B\} \in C \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by fast
  then show  $\langle Neg\ A \in ?H \rangle$  and  $\langle Neg\ B \in ?H \rangle$ 
  using  $\langle \text{maximal } ?H C \rangle$  unfolding maximal-def by fast+ }

{ fix  $A\ B$ 
  assume  $\langle Neg\ (Impl\ A\ B) \in ?H \rangle$ 
  then have  $\langle ?H \cup \{A, Neg\ B\} \in C \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by blast
  then show  $\langle A \in ?H \rangle$  and  $\langle Neg\ B \in ?H \rangle$ 
  using  $\langle \text{maximal } ?H C \rangle$  unfolding maximal-def by fast+ }

{ fix  $A\ B$ 
  assume  $\langle Or\ A\ B \in ?H \rangle$ 
  then have  $\langle ?H \cup \{A\} \in C \vee ?H \cup \{B\} \in C \rangle$ 
  using  $\langle ?H \in C \rangle$  altc unfolding alt-consistency-def by fast

```

**then show**  $\langle A \in ?H \vee B \in ?H \rangle$   
**using**  $\langle \text{maximal } ?H \ C \rangle$  **unfolding** *maximal-def* **by fast** }

{ **fix**  $A \ B$   
**assume**  $\langle \text{Neg } (And \ A \ B) \in ?H \rangle$   
**then have**  $\langle ?H \cup \{ \text{Neg } A \} \in C \vee ?H \cup \{ \text{Neg } B \} \in C \rangle$   
**using**  $\langle ?H \in C \rangle$  **altc** **unfolding** *alt-consistency-def* **by simp**  
**then show**  $\langle \text{Neg } A \in ?H \vee \text{Neg } B \in ?H \rangle$   
**using**  $\langle \text{maximal } ?H \ C \rangle$  **unfolding** *maximal-def* **by fast** }

{ **fix**  $A \ B$   
**assume**  $\langle \text{Impl } A \ B \in ?H \rangle$   
**then have**  $\langle ?H \cup \{ \text{Neg } A \} \in C \vee ?H \cup \{ B \} \in C \rangle$   
**using**  $\langle ?H \in C \rangle$  **altc** **unfolding** *alt-consistency-def* **by simp**  
**then show**  $\langle \text{Neg } A \in ?H \vee B \in ?H \rangle$   
**using**  $\langle \text{maximal } ?H \ C \rangle$  **unfolding** *maximal-def* **by fast** }

{ **fix**  $P$  **and**  $t :: \langle \text{nat term} \rangle$   
**assume**  $\langle \text{Forall } P \in ?H \rangle$  **and**  $\langle \text{closedt } 0 \ t \rangle$   
**then have**  $\langle ?H \cup \{ P[t/0] \} \in C \rangle$   
**using**  $\langle ?H \in C \rangle$  **altc** **unfolding** *alt-consistency-def* **by blast**  
**then show**  $\langle P[t/0] \in ?H \rangle$   
**using**  $\langle \text{maximal } ?H \ C \rangle$  **unfolding** *maximal-def* **by fast** }

{ **fix**  $P$  **and**  $t :: \langle \text{nat term} \rangle$   
**assume**  $\langle \text{Neg } (Exists \ P) \in ?H \rangle$  **and**  $\langle \text{closedt } 0 \ t \rangle$   
**then have**  $\langle ?H \cup \{ \text{Neg } (P[t/0]) \} \in C \rangle$   
**using**  $\langle ?H \in C \rangle$  **altc** **unfolding** *alt-consistency-def* **by blast**  
**then show**  $\langle \text{Neg } (P[t/0]) \in ?H \rangle$   
**using**  $\langle \text{maximal } ?H \ C \rangle$  **unfolding** *maximal-def* **by fast** }

{ **fix**  $P$   
**assume**  $\langle Exists \ P \in ?H \rangle$   
**obtain**  $n$  **where**  $*$ :  $\langle Exists \ P = f \ n \rangle$   
**using** *surj* **by blast**

**let**  $?t = \langle App \ (SOME \ k. \ k \notin (\bigcup p \in \text{extend } S \ C \ f \ n \cup \{ f \ n \}. \ \text{params } p)) \ [] \rangle$   
**have**  $\langle \text{closedt } 0 \ ?t \rangle$  **by simp**

**have**  $\langle Exists \ P \in (\bigcup n. \ \text{extend } S \ C \ f \ n) \rangle$   
**using**  $\langle Exists \ P \in ?H \rangle$  *Extend-def* **by blast**  
**then have**  $\langle \text{extend } S \ C \ f \ n \cup \{ f \ n \} \subseteq (\bigcup n. \ \text{extend } S \ C \ f \ n) \rangle$   
**using**  $*$  **by** (*simp add: UN-upper*)  
**then have**  $\langle \text{extend } S \ C \ f \ n \cup \{ f \ n \} \in C \rangle$   
**using** *Extend-def*  $\langle \text{Extend } S \ C \ f \in C \rangle$  *fin-ch finite-char-closed*  
**unfolding** *subset-closed-def* **by metis**  
**then have**  $\langle P[?t/0] \in \text{extend } S \ C \ f \ (Suc \ n) \rangle$   
**using**  $*$  *Exists-in-extend* **by blast**  
**then have**  $\langle P[?t/0] \in ?H \rangle$

**using** *Extend-def* **by** *blast*  
**then show**  $\langle \exists t. \text{closedt } 0 \ t \wedge P[t/0] \in ?H \rangle$   
**using**  $\langle \text{closedt } 0 \ ?t \rangle$  **by** *blast* }

{ **fix** *P*  
**assume**  $\langle \text{Neg } (\text{Forall } P) \in ?H \rangle$   
**obtain** *n* **where** \*:  $\langle \text{Neg } (\text{Forall } P) = f \ n \rangle$   
**using** *surj* **by** *blast*

**let**  $?t = \langle \text{App } (\text{SOME } k. k \notin (\bigcup p \in \text{extend } S \ C \ f \ n \cup \{f \ n\}. \text{params } p)) \ \square \rangle$   
**have**  $\langle \text{closedt } 0 \ ?t \rangle$  **by** *simp*

**have**  $\langle \text{Neg } (\text{Forall } P) \in (\bigcup n. \text{extend } S \ C \ f \ n) \rangle$   
**using**  $\langle \text{Neg } (\text{Forall } P) \in ?H \rangle$  *Extend-def* **by** *blast*  
**then have**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \subseteq (\bigcup n. \text{extend } S \ C \ f \ n) \rangle$   
**using** \* **by** (*simp add: UN-upper*)  
**then have**  $\langle \text{extend } S \ C \ f \ n \cup \{f \ n\} \in C \rangle$   
**using** *Extend-def*  $\langle \text{Extend } S \ C \ f \in C \rangle$  *fin-ch finite-char-closed*  
**unfolding** *subset-closed-def* **by** *metis*  
**then have**  $\langle \text{Neg } (P[?t/0]) \in \text{extend } S \ C \ f \ (\text{Suc } n) \rangle$   
**using** \* *Neg-Forall-in-extend* **by** *blast*  
**then have**  $\langle \text{Neg } (P[?t/0]) \in ?H \rangle$   
**using** *Extend-def* **by** *blast*  
**then show**  $\langle \exists t. \text{closedt } 0 \ t \wedge \text{Neg } (P[t/0]) \in ?H \rangle$   
**using**  $\langle \text{closedt } 0 \ ?t \rangle$  **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**  $\langle \text{consistency } C \rangle$  **and**  $\langle S \in C \rangle$   
**and**  $\langle \text{infinite } (- (\bigcup p \in S. \text{params } p)) \rangle$   
**shows**  $\langle \text{hintikka } (\text{Extend } S \ (\text{mk-finite-char } (\text{mk-alt-consistency } (\text{close } C))) \ \text{from-nat}) \rangle$   
**(is**  $\langle \text{hintikka } (\text{Extend } S \ ?C' \ \text{from-nat}) \rangle$ **)**

**proof** –

**have**  $\langle \text{finite-char } ?C' \rangle$   
**using** *finite-char* **by** *blast*  
**moreover have**  $\langle \forall y. y = \text{from-nat } (\text{to-nat } y) \rangle$   
**by** *simp*  
**then have**  $\langle \forall y. \exists n. y = \text{from-nat } n \rangle$   
**by** *blast*  
**moreover have**  $\langle \text{alt-consistency } ?C' \rangle$   
**using** *alt-consistency close-closed close-consistency*  $\langle \text{consistency } C \rangle$   
*finite-alt-consistency mk-alt-consistency-closed*  
**by** *blast*  
**moreover have**  $\langle S \in \text{close } C \rangle$   
**using** *close-subset*  $\langle S \in C \rangle$  **by** *blast*

**then have**  $\langle S \in \text{mk-alt-consistency } (\text{close } C) \rangle$   
**using** *mk-alt-consistency-subset* **by** *blast*  
**then have**  $\langle S \in ?C' \rangle$   
**using** *close-closed finite-char-subset mk-alt-consistency-closed* **by** *blast*  
**ultimately show** *?thesis*  
**using** *extend-hintikka*  $\langle \text{infinite } (- (\bigcup p \in S. \text{params } p)) \rangle$   
**by** *metis*  
**qed**

**theorem** *model-existence*:  
**assumes**  $\langle \text{consistency } C \rangle$   
**and**  $\langle S \in C \rangle$   
**and**  $\langle \text{infinite } (- (\bigcup p \in S. \text{params } p)) \rangle$   
**and**  $\langle p \in S \rangle$   
**and**  $\langle \text{closed } 0 \ p \rangle$   
**shows**  $\langle \text{eval } e \text{ HApp } (\lambda a \text{ ts. Pred } a \text{ (terms-of-hterms ts)} \in \text{Extend } S$   
 $\text{ (mk-finite-char (mk-alt-consistency (close } C))) \text{ from-nat } p) \rangle$   
**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*:  $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$   
**shows**  $\langle \text{consistency } \{S :: ('a, 'b) \text{ form set. } \exists G. S = \text{set } G \wedge \neg G \vdash FF\} \rangle$   
**unfolding** *consistency-def*  
**proof** (*intro conjI allI impI notI*)  
**fix**  $S :: \langle ('a, 'b) \text{ form set} \rangle$   
**assume**  $\langle S \in \{\text{set } G \mid G. \neg G \vdash FF\} \rangle$  (**is**  $\langle S \in ?C \rangle$ )  
**then obtain**  $G :: \langle ('a, 'b) \text{ form list} \rangle$   
**where**  $*$ :  $\langle S = \text{set } G \rangle$  **and**  $\langle \neg G \vdash FF \rangle$   
**by** *blast*  
  
**{** **fix**  $p \text{ ts}$   
**assume**  $\langle \text{Pred } p \text{ ts} \in S \wedge \text{Neg } (\text{Pred } p \text{ ts}) \in S \rangle$   
**then have**  $\langle G \vdash \text{Pred } p \text{ ts} \rangle$  **and**  $\langle G \vdash \text{Neg } (\text{Pred } p \text{ ts}) \rangle$   
**using** *Assum \** **by** *blast+*  
**then have**  $\langle G \vdash FF \rangle$   
**using** *NegE* **by** *blast*  
**then show** *False*  
**using**  $\langle \neg G \vdash FF \rangle$  **by** *blast* **}**  
  
**{** **assume**  $\langle FF \in S \rangle$   
**then have**  $\langle G \vdash FF \rangle$

```

    using Assum * by blast
  then show False
    using  $\langle \neg G \vdash FF \rangle$  by blast }

{ assume  $\langle Neg\ TT \in S \rangle$ 
  then have  $\langle G \vdash Neg\ TT \rangle$ 
    using Assum * by blast
  moreover have  $\langle G \vdash TT \rangle$ 
    using TTI by blast
  ultimately have  $\langle G \vdash FF \rangle$ 
    using NegE by blast
  then show False
    using  $\langle \neg G \vdash FF \rangle$  by blast }

{ fix Z
  assume  $\langle Neg\ (Neg\ Z) \in S \rangle$ 
  then have  $\langle G \vdash Neg\ (Neg\ Z) \rangle$ 
    using Assum * by blast

  { assume  $\langle Z \# G \vdash FF \rangle$ 
    then have  $\langle G \vdash Neg\ Z \rangle$ 
      using NegI by blast
    then have  $\langle G \vdash FF \rangle$ 
      using NegE  $\langle G \vdash Neg\ (Neg\ Z) \rangle$  by blast
    then have False
      using  $\langle \neg G \vdash FF \rangle$  by blast }
  then have  $\langle \neg Z \# G \vdash FF \rangle$ 
    by blast
  moreover have  $\langle S \cup \{Z\} = set\ (Z \# G) \rangle$ 
    using * by simp
  ultimately show  $\langle S \cup \{Z\} \in ?C \rangle$ 
    by blast }

{ fix A B
  assume  $\langle And\ A\ B \in S \rangle$ 
  then have  $\langle G \vdash And\ A\ B \rangle$ 
    using Assum * by blast
  then have  $\langle G \vdash A \rangle$  and  $\langle G \vdash B \rangle$ 
    using AndE1 AndE2 by blast+

  { assume  $\langle A \# B \# G \vdash FF \rangle$ 
    then have  $\langle B \# G \vdash Neg\ A \rangle$ 
      using NegI by blast
    then have  $\langle G \vdash Neg\ A \rangle$ 
      using cut  $\langle G \vdash B \rangle$  by blast
    then have  $\langle G \vdash FF \rangle$ 
      using NegE  $\langle G \vdash A \rangle$  by blast
    then have False
      using  $\langle \neg G \vdash FF \rangle$  by blast }

```

```

then have  $\langle \neg A \# B \# G \vdash FF \rangle$ 
  by blast
moreover have  $\langle S \cup \{A, B\} = \text{set } (A \# B \# G) \rangle$ 
  using * by simp
ultimately show  $\langle S \cup \{A, B\} \in ?C \rangle$ 
  by blast }

{ fix  $A B$ 
  assume  $\langle \text{Neg } (Or A B) \in S \rangle$ 
  then have  $\langle G \vdash \text{Neg } (Or A B) \rangle$ 
    using Assum * by blast

  have  $\langle A \# \text{Neg } B \# G \vdash A \rangle$ 
    by (simp add: Assum)
  then have  $\langle A \# \text{Neg } B \# G \vdash Or A B \rangle$ 
    using OrI1 by blast
  moreover have  $\langle A \# \text{Neg } B \# G \vdash \text{Neg } (Or A B) \rangle$ 
    using *  $\langle \text{Neg } (Or A B) \in S \rangle$  by (simp add: Assum)
  ultimately have  $\langle A \# \text{Neg } B \# G \vdash FF \rangle$ 
    using NegE  $\langle A \# \text{Neg } B \# G \vdash \text{Neg } (Or A B) \rangle$  by blast
  then have  $\langle \text{Neg } B \# G \vdash \text{Neg } A \rangle$ 
    using NegI by blast

  have  $\langle B \# G \vdash B \rangle$ 
    by (simp add: Assum)
  then have  $\langle B \# G \vdash Or A B \rangle$ 
    using OrI2 by blast
  moreover have  $\langle B \# G \vdash \text{Neg } (Or A B) \rangle$ 
    using *  $\langle \text{Neg } (Or A B) \in S \rangle$  by (simp add: Assum)
  ultimately have  $\langle B \# G \vdash FF \rangle$ 
    using NegE  $\langle B \# G \vdash \text{Neg } (Or A B) \rangle$  by blast
  then have  $\langle G \vdash \text{Neg } B \rangle$ 
    using NegI by blast

  { assume  $\langle \text{Neg } A \# \text{Neg } B \# G \vdash FF \rangle$ 
    then have  $\langle \text{Neg } B \# G \vdash \text{Neg } (\text{Neg } A) \rangle$ 
      using NegI by blast
    then have  $\langle \text{Neg } B \# G \vdash FF \rangle$ 
      using NegE  $\langle \text{Neg } B \# G \vdash \text{Neg } A \rangle$  by blast
    then have  $\langle G \vdash FF \rangle$ 
      using cut  $\langle G \vdash \text{Neg } B \rangle$  by blast
    then have False
      using  $\langle \neg G \vdash FF \rangle$  by blast }
  then have  $\langle \neg \text{Neg } A \# \text{Neg } B \# G \vdash FF \rangle$ 
    by blast
  moreover have  $\langle S \cup \{\text{Neg } A, \text{Neg } B\} = \text{set } (\text{Neg } A \# \text{Neg } B \# G) \rangle$ 
    using * by simp
  ultimately show  $\langle S \cup \{\text{Neg } A, \text{Neg } B\} \in ?C \rangle$ 
    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 ∈ S⟩
  then have ⟨G ⊢ Or A B⟩
    using * Assum by blast

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

{ fix A B
  assume ⟨Neg (And A B) ∈ S⟩

  let ?x = ⟨Or (Neg A) (Neg B)⟩

  have ⟨B # A # Neg ?x # G ⊢ A⟩ and ⟨B # A # Neg ?x # G ⊢ B⟩
    by (simp-all add: Assum)
  then have ⟨B # A # Neg ?x # G ⊢ And A B⟩
    using AndI by blast
  moreover have ⟨B # A # Neg ?x # G ⊢ Neg (And A B)⟩
    using * ⟨Neg (And A B) ∈ S⟩ by (simp add: Assum)
  ultimately have ⟨B # A # Neg ?x # G ⊢ FF⟩
    using NegE by blast
  then have ⟨A # Neg ?x # G ⊢ Neg B⟩
    using NegI 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) (Neg B)⟩
    using Class' by blast

  { assume ⟨(∀ G'. set G' = S ∪ {Neg A} ⟶ G' ⊢ FF)⟩
    and ⟨(∀ G'. set G' = S ∪ {Neg B} ⟶ G' ⊢ FF)⟩
    then have ⟨Neg A # G ⊢ FF⟩ and ⟨Neg B # G ⊢ FF⟩
      using * by simp-all
    then have ⟨G ⊢ 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  $\langle G \vdash FF \rangle$ 
  using cut  $\langle G \vdash P[t/0] \rangle$  by blast
then have False
  using  $\langle \neg G \vdash FF \rangle$  by blast }
then have  $\langle \neg P[t/0] \# G \vdash FF \rangle$ 
  by blast
moreover have  $\langle S \cup \{P[t/0]\} = \text{set } (P[t/0] \# G) \rangle$ 
  using * by simp
ultimately show  $\langle S \cup \{P[t/0]\} \in ?C \rangle$ 
  by blast }

{ fix P and t ::  $\langle 'a \text{ term} \rangle$ 
  assume  $\langle \text{closed } 0 \ t \rangle$  and  $\langle \text{Neg } (\text{Exists } P) \in S \rangle$ 
  then have  $\langle G \vdash \text{Neg } (\text{Exists } P) \rangle$ 
    using Assum * by blast
  then have  $\langle P[t/0] \in \text{set } (P[t/0] \# G) \rangle$ 
    by (simp add: Assum)
  then have  $\langle P[t/0] \# G \vdash P[t/0] \rangle$ 
    using Assum by blast
  then have  $\langle P[t/0] \# G \vdash \text{Exists } P \rangle$ 
    using ExistsI by blast
  moreover have  $\langle P[t/0] \# G \vdash \text{Neg } (\text{Exists } P) \rangle$ 
    using *  $\langle \text{Neg } (\text{Exists } P) \in S \rangle$  by (simp add: Assum)
  ultimately have  $\langle P[t/0] \# G \vdash FF \rangle$ 
    using NegE by blast
  then have  $\langle G \vdash \text{Neg } (P[t/0]) \rangle$ 
    using NegI by blast

  { assume  $\langle \text{Neg } (P[t/0]) \# G \vdash FF \rangle$ 
    then have  $\langle G \vdash FF \rangle$ 
      using cut  $\langle G \vdash \text{Neg } (P[t/0]) \rangle$  by blast
    then have False
      using  $\langle \neg G \vdash FF \rangle$  by blast }
  then have  $\langle \neg \text{Neg } (P[t/0]) \# G \vdash FF \rangle$ 
    by blast
  moreover have  $\langle S \cup \{\text{Neg } (P[t/0])\} = \text{set } (\text{Neg } (P[t/0]) \# G) \rangle$ 
    using * by simp
  ultimately show  $\langle S \cup \{\text{Neg } (P[t/0])\} \in ?C \rangle$ 
    by blast }

{ fix P
  assume  $\langle \text{Exists } P \in S \rangle$ 
  then have  $\langle G \vdash \text{Exists } P \rangle$ 
    using * Assum by blast

  have  $\langle \text{finite } ((\bigcup p \in \text{set } G. \text{params } p) \cup \text{params } P) \rangle$ 
    by simp
  then have  $\langle \text{infinite } (\neg ((\bigcup p \in \text{set } G. \text{params } p) \cup \text{params } P)) \rangle$ 
    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 ( $\lambda p. x \notin \text{params } p$ ) ( $?x \# G$ )⟩
    using ** by (simp add: list-all-iff)
  moreover have ⟨ $x \notin \text{params } P$ ⟩
    using ** by simp
  ultimately have ⟨ $?x \# G \vdash \text{Forall } P$ ⟩
    using ForallI by fast
  moreover have ⟨ $?x \# G \vdash \text{Neg } (\text{Forall } P)$ ⟩
    using * ⟨ $\text{Neg } (\text{Forall } P) \in S$ ⟩ by (simp add: Assum)
  ultimately have ⟨ $?x \# G \vdash FF$ ⟩
    using NegE by blast
  then have ⟨ $G \vdash \text{Exists } (\text{Neg } P)$ ⟩
    using Class by blast

{ assume ⟨ $\text{Neg } (P[\text{App } x \ []/0]) \# G \vdash FF$ ⟩
  moreover have ⟨list-all ( $\lambda p. x \notin \text{params } p$ )  $G$ ⟩
    using ** by (simp add: list-all-iff)
  moreover have ⟨ $x \notin \text{params } P$ ⟩
    using ** by simp
  moreover have ⟨ $x \notin \text{params } FF$ ⟩
    by simp
  ultimately have ⟨ $G \vdash FF$ ⟩
    using ExistsE ⟨ $G \vdash \text{Exists } (\text{Neg } P)$ ⟩ by fastforce
  then have False
    using ⟨ $\neg G \vdash FF$ ⟩
    by blast }
  then have ⟨ $\neg \text{Neg } (P[\text{App } x \ []/0]) \# G \vdash FF$ ⟩
    by blast
  moreover have ⟨ $S \cup \{\text{Neg } (P[\text{App } x \ []/0])\} = \text{set } (\text{Neg } (P[\text{App } x \ []/0]) \# G)$ ⟩
    using * by simp
  ultimately show ⟨ $\exists x. S \cup \{\text{Neg } (P[\text{App } x \ []/0])\} \in ?C$ ⟩
    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: ⟨ $\forall e f g. e, (f :: \text{nat} \Rightarrow \text{nat hterm list} \Rightarrow \text{nat hterm}),$ 
      ( $g :: \text{nat} \Rightarrow \text{nat hterm list} \Rightarrow \text{bool}$ ),  $ps \models p$ ⟩
  shows ⟨ $ps \vdash p$ ⟩
proof (rule Class, rule ccontr)
  fix  $e$ 
  assume ⟨ $\neg \text{Neg } p \# ps \vdash FF$ ⟩

```

```

  let  $?S = \langle \text{set } (\text{Neg } p \# ps) \rangle$ 
  let  $?C = \langle \{ \text{set } (G :: (\text{nat}, \text{nat}) \text{ form list}) \mid G. \neg G \vdash FF \} \rangle$ 
  let  $?f = \text{HApp}$ 
  let  $?g = \langle \lambda a ts. \text{Pred } a \text{ (terms-of-hterms } ts) \in \text{Extend } ?S \rangle$ 

```

(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  $*$ :  $\langle \forall x \in S. \text{eval } e f g x \rangle$  by *blast*

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

```
show  $\langle FF \notin S \rangle$ 
  using  $*$  by fastforce
```

```
show  $\langle Neg TT \notin S \rangle$ 
  using  $*$  by fastforce
```

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

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

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

```

{ fix A B
  assume ⟨Neg (Impl A B) ∈ S⟩
  then have ⟨∀ x ∈ S ∪ {Neg (Impl A B)}. eval e f g x⟩
    using * by blast
  then have ⟨∀ x ∈ S ∪ {A, Neg B}. eval e f g x⟩
    by simp
  moreover have ⟨infinite (− (∪ p ∈ S ∪ {A, Neg B}. params p))⟩
    using inf-params by simp
  ultimately show ⟨S ∪ {A, Neg B} ∈ ?C⟩
    by blast }

{ fix A B
  assume ⟨Or A B ∈ S⟩
  then have ⟨∀ x ∈ S ∪ {Or A B}. eval e f g x⟩
    using * by blast
  then have ⟨(∀ x ∈ S ∪ {A}. eval e f g x) ∨
    (∀ x ∈ S ∪ {B}. eval e f g x)⟩
    by simp
  moreover have ⟨infinite (− (∪ p ∈ S ∪ {A}. params p))⟩
    and ⟨infinite (− (∪ p ∈ S ∪ {B}. params p))⟩
    using inf-params by simp-all
  ultimately show ⟨S ∪ {A} ∈ ?C ∨ S ∪ {B} ∈ ?C⟩
    by blast }

{ fix A B
  assume ⟨Neg (And A B) ∈ S⟩
  then have ⟨∀ x ∈ S ∪ {Neg (And A B)}. eval e f g x⟩
    using * by blast
  then have ⟨(∀ x ∈ S ∪ {Neg A}. eval e f g x) ∨
    (∀ x ∈ S ∪ {Neg B}. eval e f g x)⟩
    by simp
  moreover have ⟨infinite (− (∪ p ∈ S ∪ {Neg A}. params p))⟩
    and ⟨infinite (− (∪ p ∈ S ∪ {Neg B}. params p))⟩
    using inf-params by simp-all
  ultimately show ⟨S ∪ {Neg A} ∈ ?C ∨ S ∪ {Neg B} ∈ ?C⟩
    by blast }

{ fix A B
  assume ⟨Impl A B ∈ S⟩
  then have ⟨∀ x ∈ S ∪ {Impl A B}. eval e f g x⟩
    using * by blast
  then have ⟨(∀ x ∈ S ∪ {Neg A}. eval e f g x) ∨
    (∀ x ∈ S ∪ {B}. eval e f g x)⟩
    by simp
  moreover have ⟨infinite (− (∪ p ∈ S ∪ {Neg A}. params p))⟩
    and ⟨infinite (− (∪ p ∈ S ∪ {B}. params p))⟩
    using inf-params by simp-all
  ultimately show ⟨S ∪ {Neg A} ∈ ?C ∨ S ∪ {B} ∈ ?C⟩

```



```

    by blast }

{ fix P and t :: <'a term>
  assume <Forall P ∈ S>
  then have <∀ x ∈ S ∪ {Forall P}. eval e f g x>
    using * by blast
  then have <∀ x ∈ S ∪ {P[t/0]}. eval e f g x>
    by simp
  moreover have <infinite (− (∪ p ∈ S ∪ {P[t/0]}. params p))>
    using inf-params by simp
  ultimately show <S ∪ {P[t/0]} ∈ ?C>
    by blast }

{ fix P and t :: <'a term>
  assume <Neg (Exists P) ∈ S>
  then have <∀ x ∈ S ∪ {Neg (Exists P)}. eval e f g x>
    using * by blast
  then have <∀ x ∈ S ∪ {Neg (P[t/0])}. eval e f g x>
    by simp
  moreover have <infinite (− (∪ p ∈ S ∪ {Neg (P[t/0])}. params p))>
    using inf-params by simp
  ultimately show <S ∪ {Neg (P[t/0])} ∈ ?C>
    by blast }

{ fix P
  assume <Exists P ∈ S>
  then have <∀ x ∈ S ∪ {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⟨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 <Exists 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 ∪ {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 ∪ {P[App x []/0]} ∈ ?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 (λn. 2 * n) ‘ S. eval e (λn. f (n div 2)) g p⟩
  by (simp add: psubst-eval)
  then have ⟨psubst (λn. 2 * n) ‘ S ∈ ?C⟩
  using doublep-infinite-params by blast
  moreover have ⟨psubst (λn. 2 * n) p ∈ psubst (λn. 2 * n) ‘ S⟩
  using ⟨p ∈ S⟩ by blast
  moreover have ⟨closed 0 (psubst (λn. 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 (λn. 2 * n) p)⟩
  using model-existence by blast
  then show ⟨eval e' (λn. 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**  $\langle \text{news fresh } (\text{map } (\text{psubst } ?f) (p \# z)) \rangle$   
 using  $\langle \text{new fresh } (\text{psubst } ?f p) \rangle \langle \text{news fresh } (\text{map } (\text{psubst } ?f) z) \rangle$   
 by *simp*  
**then have**  $\langle \text{new fresh } (\text{psubst } ?f p) \rangle \langle \text{news fresh } (\text{map } (\text{psubst } ?f) z) \rangle$   
 by *simp-all*  
**ultimately have**  $\langle \text{map } (\text{psubst } ?f) z \vdash \text{Forall } (\text{psubst } ?f p) \rangle$   
 using *deriv.ForallI* by *metis*  
**then show** *?case*  
 using *ForallI*  $\langle \text{map } (\text{psubst } ?f) z = \text{map } (\text{psubst } f) z \rangle$  by *simp*  
**qed**

## 9.2 Substitution for constants

**primrec**

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

**primrec**  $\text{subc} :: \langle 'a \Rightarrow 'a \text{ term} \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$  **where**

$\langle \text{subc } c s FF = FF \rangle \mid$   
 $\langle \text{subc } c s TT = TT \rangle \mid$   
 $\langle \text{subc } c s (\text{Pred } i l) = \text{Pred } i (\text{subc-list } c s l) \rangle \mid$   
 $\langle \text{subc } c s (\text{Neg } p) = \text{Neg } (\text{subc } c s p) \rangle \mid$   
 $\langle \text{subc } c s (\text{Impl } p q) = \text{Impl } (\text{subc } c s p) (\text{subc } c s q) \rangle \mid$   
 $\langle \text{subc } c s (\text{Or } p q) = \text{Or } (\text{subc } c s p) (\text{subc } c s q) \rangle \mid$   
 $\langle \text{subc } c s (\text{And } p q) = \text{And } (\text{subc } c s p) (\text{subc } c s q) \rangle \mid$   
 $\langle \text{subc } c s (\text{Exists } p) = \text{Exists } (\text{subc } c (\text{liftt } s) p) \rangle \mid$   
 $\langle \text{subc } c s (\text{Forall } p) = \text{Forall } (\text{subc } c (\text{liftt } s) p) \rangle$

**primrec**  $\text{subcs} :: \langle 'a \Rightarrow 'a \text{ term} \Rightarrow ('a, 'b) \text{ form list} \Rightarrow ('a, 'b) \text{ form list} \rangle$  **where**

$\langle \text{subcs } c s [] = [] \rangle \mid$   
 $\langle \text{subcs } c s (p \# z) = \text{subc } c s p \# \text{subcs } c s z \rangle$

**lemma** *subst-0-lift*:

$\langle \text{substt } (\text{liftt } t) s 0 = t \rangle$   
 $\langle \text{substts } (\text{liftts } l) s 0 = l \rangle$   
**by** (*induct t and l rule: substt.induct substts.induct*) *simp-all*

**lemma** *params-lift* [*simp*]:

**fixes**  $t :: \langle 'a \text{ term} \rangle$  **and**  $ts :: \langle 'a \text{ term list} \rangle$   
**shows**  
 $\langle \text{paramst } (\text{liftt } t) = \text{paramst } t \rangle$   
 $\langle \text{paramsts } (\text{liftts } ts) = \text{paramsts } ts \rangle$   
**by** (*induct t and ts rule: paramst.induct paramsts.induct*) *simp-all*

**lemma** *subst-new'* [simp]:

$\langle \text{new-term } c \ s \implies \text{new-term } c \ t \implies \text{new-term } c \ (\text{substt } t \ s \ m) \rangle$

$\langle \text{new-term } c \ s \implies \text{new-list } c \ l \implies \text{new-list } c \ (\text{substts } l \ s \ m) \rangle$

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

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

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

**lemma** *subst-new-all*:

**assumes**  $\langle a \notin \text{set } cs \rangle \langle \text{list-all } (\lambda c. \text{new } c \ p) \ cs \rangle$

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

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

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

$\langle \text{new-term } c \ t \implies \text{subc-term } c \ s \ t = t \rangle$

$\langle \text{new-list } c \ l \implies \text{subc-list } c \ s \ l = l \rangle$

**by** (induct *t* and *l* rule: *subc-term.induct* *subc-list.induct*) *auto*

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

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

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

**by** (induct *z*) *simp-all*

**lemma** *subc-psubst'* [simp]:

$\langle (\forall x \in \text{paramst } t. x \neq c \longrightarrow f \ x \neq f \ c) \implies$

$\text{psubstt } f \ (\text{subc-term } c \ s \ t) = \text{subc-term } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubstt } f \ t) \rangle$

$\langle (\forall x \in \text{paramsts } l. x \neq c \longrightarrow f \ x \neq f \ c) \implies$

$\text{psubstts } f \ (\text{subc-list } c \ s \ l) = \text{subc-list } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubstts } f \ l) \rangle$

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

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

$\text{psubst } f \ (\text{subc } c \ s \ p) = \text{subc } (f \ c) \ (\text{psubstt } f \ s) \ (\text{psubst } f \ p) \rangle$

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

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

$\text{map } (\text{psubst } f) \ (\text{subcs } c \ s \ z) = \text{subcs } (f \ c) \ (\text{psubstt } f \ s) \ (\text{map } (\text{psubst } f) \ z) \rangle$

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

**lemma** *new-lift*:

$\langle \text{new-term } c \ t \implies \text{new-term } c \ (\text{liftt } t) \rangle$

$\langle \text{new-list } c \ l \implies \text{new-list } c \ (\text{liftts } l) \rangle$

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

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

$\langle \text{new-term } d \ s \implies \text{new-term } d \ t \implies \text{new-term } d \ (\text{subc-term } c \ s \ t) \rangle$

$\langle \text{new-term } d \ s \implies \text{new-list } d \ l \implies \text{new-list } d \ (\text{subc-list } c \ s \ l) \rangle$

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



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

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

**lemma** *psubst-new-free'*:  
 $\langle c \neq n \implies \text{new-term } n \ (\text{psubstt } (\text{id}(n := c)) \ t) \rangle$   
 $\langle c \neq n \implies \text{new-list } n \ (\text{psubstts } (\text{id}(n := c)) \ l) \rangle$   
**by** (induct *t* and *l* rule: paramst.induct paramsts.induct) simp-all

**lemma** *psubst-new-free*:  $\langle c \neq n \implies \text{new } n \ (\text{psubst } (\text{id}(n := c)) \ p) \rangle$   
**using** *psubst-new-free'* **by** (induct *p*) fastforce+

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

**lemma** *psubst-new-away'* [simp]:  
 $\langle \text{new-term } \text{fresh } t \implies \text{psubstt } (\text{id}(\text{fresh} := c)) \ (\text{psubstt } (\text{id}(c := \text{fresh})) \ t) = t \rangle$   
 $\langle \text{new-list } \text{fresh } l \implies \text{psubstts } (\text{id}(\text{fresh} := c)) \ (\text{psubstts } (\text{id}(c := \text{fresh})) \ l) = l \rangle$   
**by** (induct *t* and *l* rule: psubstt.induct psubstts.induct) auto

**lemma** *psubst-new-away* [simp]:  $\langle \text{new } \text{fresh } p \implies \text{psubst } (\text{id}(\text{fresh} := c)) \ (\text{psubst } (\text{id}(c := \text{fresh})) \ p) = p \rangle$   
**by** (induct *p*) simp-all

**lemma** *map-psubst-new-away*:  
 $\langle \text{news } \text{fresh } z \implies \text{map } (\text{psubst } (\text{id}(\text{fresh} := c))) \ (\text{map } (\text{psubst } (\text{id}(c := \text{fresh}))) \ z) = z \rangle$   
**by** (induct *z*) simp-all

**lemma** *psubst-new'*:  
 $\langle \text{new-term } c \ t \implies \text{psubstt } (\text{id}(c := x)) \ t = t \rangle$   
 $\langle \text{new-list } c \ l \implies \text{psubstts } (\text{id}(c := x)) \ l = l \rangle$   
**by** (induct *t* and *l* rule: psubstt.induct psubstts.induct) auto

**lemma** *psubst-new*:  $\langle \text{new } c \ p \implies \text{psubst } (\text{id}(c := x)) \ p = p \rangle$   
**using** *psubst-new'* **by** (induct *p*) fastforce+

**lemma** *map-psubst-new*:  $\langle \text{news } c \ z \implies \text{map } (\text{psubst } (\text{id}(c := x))) \ z = z \rangle$   
**using** *psubst-new* **by** (induct *z*) auto

**lemma** *lift-subst* [simp]:  
 $\langle \text{liftt } (\text{substt } t \ u \ m) = \text{substt } (\text{liftt } t) \ (\text{liftt } u) \ (m + 1) \rangle$   
 $\langle \text{liftts } (\text{substts } l \ u \ m) = \text{substts } (\text{liftts } l) \ (\text{liftt } u) \ (m + 1) \rangle$   
**by** (induct *t* and *l* rule: substt.induct substts.induct) simp-all

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

$\langle \text{new-term } c \ s \implies \text{new-list } c \ (\text{subc-list } c \ s \ l) \rangle$   
**by** (induct  $t$  and  $l$  rule: *subc-term.induct subc-list.induct*) *simp-all*

**lemma** *new-subc-same*:  $\langle \text{new-term } c \ s \implies \text{new } c \ (\text{subc } c \ s \ p) \rangle$   
**by** (induct  $p$  arbitrary:  $s$ ) *simp-all*

**lemma** *lift-subc*:  
 $\langle \text{liftt } (\text{subc-term } c \ s \ t) = \text{subc-term } c \ (\text{liftt } s) \ (\text{liftt } t) \rangle$   
 $\langle \text{liftts } (\text{subc-list } c \ s \ l) = \text{subc-list } c \ (\text{liftt } s) \ (\text{liftts } l) \rangle$   
**by** (induct  $t$  and  $l$  rule: *liftt.induct liftts.induct*) *simp-all*

**lemma** *new-subc-put'*:  
 $\langle \text{new-term } c \ s \implies \text{subc-term } c \ s \ (\text{substt } t \ u \ m) = \text{subc-term } c \ s \ (\text{substt } t \ (\text{subc-term } c \ s \ u) \ m) \rangle$   
 $\langle \text{new-term } c \ s \implies \text{subc-list } c \ s \ (\text{substts } l \ u \ m) = \text{subc-list } c \ s \ (\text{substts } l \ (\text{subc-term } c \ s \ u) \ m) \rangle$   
**by** (induct  $t$  and  $l$  rule: *subc-term.induct subc-list.induct*) *simp-all*

**lemma** *new-subc-put*:  
 $\langle \text{new-term } c \ s \implies \text{subc } c \ s \ (\text{subst } p \ t \ m) = \text{subc } c \ s \ (\text{subst } p \ (\text{subc-term } c \ s \ t) \ m) \rangle$

**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) (substts l u m) = substts (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  $\langle c = d \rangle$ )
  case True
  then have  $\langle z \vdash q \rangle$ 
    using ExistsE deriv.ExistsE by fast
  moreover have  $\langle \text{new } c \ q \rangle$  and  $\langle \text{news } c \ z \rangle$ 
    using ExistsE True by simp-all
  ultimately show ?thesis
    using subc-new subcs-news by metis
next
  case False
  let  $?params = \langle \text{params } p \cup \text{params } q \cup (\bigcup p \in \text{set } z. \text{params } p) \cup \text{paramst } s \cup \{c\} \cup \{d\} \rangle$ 

  have  $\langle \text{finite } ?params \rangle$ 
    by simp
  then obtain fresh where fresh:  $\langle \text{fresh} \notin ?params \rangle$ 
    using inf-params by (meson ex-new-if-finite infinite-UNIV-listI)

  let  $?s = \langle \text{psubstt } (id(d := \text{fresh})) \ s \rangle$ 
  let  $?f = \langle id(d := \text{fresh}, \text{fresh} := d) \rangle$ 

  have  $f: \langle \forall x \in ?params. x \neq c \longrightarrow ?f \ x \neq ?f \ c \rangle$ 
    using fresh by simp

  have  $\langle \text{new-term } d \ ?s \rangle$ 
    using fresh psubst-new-free'(1) by fast
  then have  $\langle \text{psubstt } ?f \ ?s = \text{psubstt } (id(\text{fresh} := d)) \ ?s \rangle$ 
    by (metis fun-upd-twist psubstt-upd(1))
  then have  $\text{psubst-s}: \langle \text{psubstt } ?f \ ?s = s \rangle$ 
    using fresh by simp

  have  $\langle ?f \ c = c \rangle$  and  $\langle \text{new-term } (?f \ c) \ (\text{App } \text{fresh} \ \square) \rangle$ 
    using False fresh by auto

  have  $\langle \text{subcs } c \ (\text{psubstt } ?f \ ?s) \ z \vdash \text{subc } c \ (\text{psubstt } ?f \ ?s) \ (\text{Exists } p) \rangle$ 
    using ExistsE by blast
  then have exi-p:
     $\langle \text{subcs } c \ s \ z \vdash \text{Exists } (\text{subc } c \ (\text{liftt } (\text{psubstt } ?f \ ?s)) \ p) \rangle$ 
    using psubst-s by simp

  have  $\langle \text{news } d \ z \rangle$ 
    using ExistsE by simp
  moreover have  $\langle \text{news } \text{fresh} \ z \rangle$ 
    using fresh by (induct z) simp-all
  ultimately have  $\langle \text{map } (\text{psubst } ?f) \ z = z \rangle$ 
    by (induct z) simp-all
  moreover have  $\langle \forall x \in \bigcup p \in \text{set } z. \text{params } p. x \neq c \longrightarrow ?f \ x \neq ?f \ c \rangle$ 

```

```

    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 =  $\langle id(c := fresh) \rangle$ 
let ?g =  $\langle id(fresh := c) \rangle$ 
let ?s =  $\langle psubstt ?f s \rangle$ 

have c:  $\langle ?g c = c \rangle$ 
  using fresh by simp
have s:  $\langle psubstt ?g ?s = s \rangle$ 
  using fresh by simp
have p:  $\langle psubst ?g (Exists p) = Exists p \rangle$ 
  using fresh by simp

have  $\langle \forall x \in (\bigcup p \in set z. params p). x \neq c \longrightarrow ?g x \neq ?g c \rangle$ 
  using fresh by auto
moreover have  $\langle map (psubst ?g) z = z \rangle$ 
  using fresh by (induct z) simp-all
ultimately have z:  $\langle map (psubst ?g) (subcs c ?s z) = subcs c s z \rangle$ 
  using s by (simp add: subcs-psubst)

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

then have  $\langle subcs c ?s z \vdash subc c ?s (Exists p) \rangle$ 
  using deriv.ExistsI by simp
then have  $\langle map (psubst ?g) (subcs c ?s z) \vdash psubst ?g (subc c ?s (Exists p)) \rangle$ 
  using deriv-psubst inf-params by blast
moreover have  $\langle \forall x \in params (Exists p). x \neq c \longrightarrow ?g x \neq ?g c \rangle$ 
  using fresh by auto
ultimately show  $\langle subcs c s z \vdash subc c s (Exists p) \rangle$ 
  using c s p z by (simp add: subc-psubst)
next
  case (ForallE z p t)
  let ?params =  $\langle params p \cup (\bigcup p \in set z. params p) \cup paramst s \cup paramst t \cup \{c\} \rangle$ 

  have  $\langle finite ?params \rangle$ 
    by simp
  then obtain fresh where fresh:  $\langle fresh \notin ?params \rangle$ 
    using inf-params by (meson ex-new-if-finite infinite-UNIV-listI)

  let ?f =  $\langle id(c := fresh) \rangle$ 
  let ?g =  $\langle id(fresh := c) \rangle$ 

```

```

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 = ⟨psubst (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 ⟨psubst ?f ?s = psubst (id(fresh := d)) ?s⟩
  by (metis fun-upd-twist psubst-upd(1))
then have psubst-s: ⟨psubst ?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) (psubst ?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 (lift 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  $\langle p \# z \vdash q \rangle$   
 using *ImplE* by *blast*  
 qed

**lemma** *remove-imps*:  
 assumes  $\langle \text{infinite } (- \text{ params } p) \rangle$   
 shows  $\langle z' \vdash \text{put-imps } p \ z \implies \text{rev } z \ @ \ z' \vdash p \rangle$   
 using *assms shift-imp-assum* by (*induct z arbitrary: z'*) *auto*

## 9.5 Closure elimination

**lemma** *subc-sub-closed-var'* [*simp*]:  
 $\langle \text{new-term } c \ t \implies \text{closedt } (Suc \ m) \ t \implies \text{subc-term } c \ (Var \ m) \ (\text{subst } t \ (App \ c \ [])) \ m) = t \rangle$   
 $\langle \text{new-list } c \ l \implies \text{closedts } (Suc \ m) \ l \implies \text{subc-list } c \ (Var \ m) \ (\text{substts } l \ (App \ c \ [])) \ m) = l \rangle$   
 by (*induct t and l rule: subst.induct substts.induct*) *auto*

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

**primrec** *put-unis* ::  $\langle \text{nat} \Rightarrow ('a, 'b) \text{ form} \Rightarrow ('a, 'b) \text{ form} \rangle$  **where**  
 $\langle \text{put-unis } 0 \ p = p \rangle \mid$   
 $\langle \text{put-unis } (Suc \ m) \ p = \text{Forall } (\text{put-unis } m \ p) \rangle$

**lemma** *sub-put-unis* [*simp*]:  
 $\langle \text{subst } (\text{put-unis } k \ p) \ (App \ c \ []) \ i = \text{put-unis } k \ (\text{subst } p \ (App \ c \ []) \ (i + k)) \rangle$   
 by (*induct k arbitrary: i*) *simp-all*

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

**lemma** *valid-put-unis*:  $\langle \forall (e :: \text{nat} \Rightarrow 'a) \ f \ g. \ \text{eval } e \ f \ g \ p \implies \text{eval } (e :: \text{nat} \Rightarrow 'a) \ f \ g \ (\text{put-unis } m \ p) \rangle$   
 by (*induct m arbitrary: e*) *simp-all*

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

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

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

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

```

then have  $\langle [] \vdash \text{Forall } (\text{put-unis } (\text{length } cs) p) \rangle$ 
  by simp
then have  $\langle [] \vdash \text{subst } (\text{put-unis } (\text{length } cs) p) (\text{App } c []) 0 \rangle$ 
  using ForallE by blast
then show ?case
  using Cons by simp
qed simp

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

lemma vars-for-consts:
  assumes  $\langle \text{infinite } (- \text{ params } p) \rangle$ 
  shows  $\langle [] \vdash p \implies [] \vdash \text{vars-for-consts } p xs \rangle$ 
  using assms deriv-subc by (induct xs arbitrary: p) fastforce+

lemma vars-for-consts-for-unis:
   $\langle \text{closed } (\text{length } cs) p \implies \text{list-all } (\lambda c. \text{new } c p) cs \implies \text{distinct } cs \implies$ 
   $\text{vars-for-consts } (\text{consts-for-unis } (\text{put-unis } (\text{length } cs) p) cs) cs = p \rangle$ 
  by (induct cs arbitrary: p) (simp-all add: subst-new-all)

lemma fresh-constant:
  fixes  $p :: \langle ('a, 'b) \text{ form} \rangle$ 
  assumes  $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$ 
  shows  $\langle \exists c. c \notin \text{set } cs \wedge \text{new } c p \rangle$ 
proof –
  have  $\langle \text{finite } (\text{set } cs \cup \text{params } p) \rangle$ 
  by simp
  then show ?thesis
  using assms ex-new-if-finite UnI1 UnI2 by metis
qed

lemma fresh-constants:
  fixes  $p :: \langle ('a, 'b) \text{ form} \rangle$ 
  assumes  $\langle \text{infinite } (\text{UNIV} :: 'a \text{ set}) \rangle$ 
  shows  $\langle \exists cs. \text{length } cs = m \wedge \text{list-all } (\lambda c. \text{new } c p) cs \wedge \text{distinct } cs \rangle$ 
proof (induct m)
  case (Suc m)
  then obtain  $cs$  where  $\langle \text{length } cs = m \wedge \text{list-all } (\lambda c. \text{new } c p) cs \wedge \text{distinct } cs \rangle$ 
  by blast
  moreover obtain  $c$  where  $\langle c \notin \text{set } cs \wedge \text{new } c p \rangle$ 
  using Suc assms fresh-constant by blast
  ultimately have  $\langle \text{length } (c \# cs) = \text{Suc } m \wedge \text{list-all } (\lambda c. \text{new } c p) (c \# cs) \wedge$ 
   $\text{distinct } (c \# cs) \rangle$ 
  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 *:  $\langle \forall (e :: \text{nat} \Rightarrow \text{nat hterm}) f g. \text{eval } e f g ?p \rangle$ 
  using assms semantics-put-imps unfolding model-def by fastforce
obtain m where **:  $\langle \text{closed } 0 (\text{put-unis } m ?p) \rangle$ 
  using ex-closure by blast
moreover have  $\langle \text{list-all } (\text{closed } 0) [] \rangle$ 
  by simp
moreover have  $\langle \forall (e :: \text{nat} \Rightarrow \text{nat hterm}) f g. e, f, g, [] \models \text{put-unis } m ?p \rangle$ 
  using * valid-put-unis unfolding model-def by blast
ultimately have  $\langle [] \vdash \text{put-unis } m ?p \rangle$ 
  using natded-complete by blast
then have  $\langle [] \vdash ?p \rangle$ 
  using ** remove-unis-sentence by fastforce
then show  $\langle z \vdash p \rangle$ 
  using remove-imps by fastforce
qed

```

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

**proposition**

```

fixes p ::  $\langle (\text{nat}, \text{nat}) \text{ form} \rangle$ 
shows  $\langle \text{valid } p \Longrightarrow \text{eval } e f g p \rangle$ 
using completeness correctness
unfolding model-def by (metis list.pred-inject(1))

```

**proposition**

```

fixes p ::  $\langle (\text{nat}, \text{nat}) \text{ form} \rangle$ 
shows  $\langle ([] \vdash p) = \text{valid } p \rangle$ 
using completeness correctness
unfolding model-def by fastforce

```

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

```

   $e, f, g, ps \models p \Longrightarrow ps \vdash p \rangle$ 
by (rule completeness)

```

## References

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