

The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

December 14, 2021

Abstract

This theory is a formalization of the resolution calculus for first-order logic. It is proven sound and complete. The soundness proof uses the substitution lemma, which shows a correspondence between substitutions and updates to an environment. The completeness proof uses semantic trees, i.e. trees whose paths are partial Herbrand interpretations. It employs Herbrand's theorem in a formulation which states that an unsatisfiable set of clauses has a finite closed semantic tree. It also uses the lifting lemma which lifts resolution derivation steps from the ground world up to the first-order world. The theory is presented in a paper in the Journal of Automated Reasoning [7] which extends a paper presented at the International Conference on Interactive Theorem Proving [6]. An earlier version was presented in an MSc thesis [5]. The formalization mostly follows textbooks by Ben-Ari [1], Chang and Lee [2], and Leitsch [4]. The theory is part of the IsaFoL project [3].

Contents

1	Terms and Literals	3
1.1	Ground	3
1.2	Auxiliary	4
1.3	Conversions	4
1.3.1	Conversions - Terms and Herbrand Terms	4
1.3.2	Conversions - Literals and Herbrand Literals	5
1.3.3	Conversions - Atoms and Herbrand Atoms	5
1.4	Enumerations	6
1.4.1	Enumerating Strings	6
1.4.2	Enumerating Herbrand Atoms	6
1.4.3	Enumerating Ground Atoms	7
2	Trees	7
2.1	Sizes	8
2.2	Paths	8
2.3	Branches	9

2.4	Internal Paths	9
2.5	Deleting Nodes	10
3	Possibly Infinite Trees	12
3.1	Infinite Paths	12
4	König's Lemma	13
5	More Terms and Literals	13
6	Clauses	14
7	Semantics	15
7.1	Semantics of Ground Terms	15
8	Substitutions	16
8.1	The Empty Substitution	17
8.2	Substitutions and Ground Terms	17
8.3	Composition	18
8.4	Merging substitutions	19
8.5	Standardizing apart	19
9	Unifiers	20
9.1	Most General Unifiers	21
10	Resolution	21
11	Soundness	22
12	Herbrand Interpretations	24
13	Partial Interpretations	24
14	Semantic Trees	26
15	Herbrand's Theorem	26
16	Lifting Lemma	28
17	Completeness	29
18	Examples	31
19	The Unification Theorem	34
20	Instance of completeness theorem	36

1 Terms and Literals

theory *TermsAndLiterals* **imports** *Main HOL-Library.Countable-Set* **begin**

type-synonym *var-sym* = *string*

type-synonym *fun-sym* = *string*

type-synonym *pred-sym* = *string*

datatype *fterm* =

Fun fun-sym (get-sub-terms: fterm list)
| *Var var-sym*

datatype *hterm* = *HFun fun-sym hterm list* — Herbrand terms defined as in Berghofer's FOL-Fitting

type-synonym *'t atom* = *pred-sym * 't list*

datatype *'t literal* =

sign: Pos (get-pred: pred-sym) (get-terms: 't list)
| *Neg (get-pred: pred-sym) (get-terms: 't list)*

fun *get-atom* :: *'t literal* \Rightarrow *'t atom* **where**

get-atom (Pos p ts) = (p, ts)
| *get-atom (Neg p ts) = (p, ts)*

1.1 Ground

fun *ground_t* :: *fterm* \Rightarrow *bool* **where**

ground_t (Var x) \longleftrightarrow False
| *ground_t (Fun f ts) \longleftrightarrow ($\forall t \in \text{set } ts. \text{ground}_t t$)*

abbreviation *ground_{ts}* :: *fterm list* \Rightarrow *bool* **where**

ground_{ts} ts \equiv ($\forall t \in \text{set } ts. \text{ground}_t t$)

abbreviation *ground_l* :: *fterm literal* \Rightarrow *bool* **where**

ground_l l \equiv ground_{ts} (get-terms l)

abbreviation *ground_{ls}* :: *fterm literal set* \Rightarrow *bool* **where**

ground_{ls} C \equiv ($\forall l \in C. \text{ground}_l l$)

definition *ground-fatoms* :: *fterm atom set* **where**

ground-fatoms \equiv {a. ground_{ts} (snd a)}

lemma *ground_l-ground-fatom*:

assumes *ground_l l*

shows *get-atom l \in ground-fatoms*

<proof>

1.2 Auxiliary

lemma *infinity*:

assumes *inj*: $\forall n :: \text{nat. } \text{undiago } (\text{diago } n) = n$

assumes *all-tree*: $\forall n :: \text{nat. } (\text{diago } n) \in S$

shows $\neg \text{finite } S$

<proof>

lemma *inv-into-f-f*:

assumes *bij-betw* $f A B$

assumes $a \in A$

shows $(\text{inv-into } A f) (f a) = a$

<proof>

lemma *f-inv-into-f*:

assumes *bij-betw* $f A B$

assumes $b \in B$

shows $f ((\text{inv-into } A f) b) = b$

<proof>

1.3 Conversions

1.3.1 Conversions - Terms and Herbrand Terms

fun *fterm-of-hterm* :: $\text{hterm} \Rightarrow \text{fterm}$ **where**

fterm-of-hterm $(\text{HFun } p \text{ ts}) = \text{Fun } p (\text{map } \text{fterm-of-hterm } \text{ts})$

definition *fterms-of-hterms* :: $\text{hterm list} \Rightarrow \text{fterm list}$ **where**

fterms-of-hterms $\text{ts} \equiv \text{map } \text{fterm-of-hterm } \text{ts}$

fun *hterm-of-fterm* :: $\text{fterm} \Rightarrow \text{hterm}$ **where**

hterm-of-fterm $(\text{Fun } p \text{ ts}) = \text{HFun } p (\text{map } \text{hterm-of-fterm } \text{ts})$

definition *hterms-of-fterms* :: $\text{fterm list} \Rightarrow \text{hterm list}$ **where**

hterms-of-fterms $\text{ts} \equiv \text{map } \text{hterm-of-fterm } \text{ts}$

lemma *hterm-of-fterm-fterm-of-hterm[simp]*: $\text{hterm-of-fterm } (\text{fterm-of-hterm } t) = t$

<proof>

lemma *hterms-of-fterms-fterms-of-hterms[simp]*: $\text{hterms-of-fterms } (\text{fterms-of-hterms } \text{ts}) = \text{ts}$

<proof>

lemma *fterm-of-hterm-hterm-of-fterm[simp]*:

assumes $\text{ground}_t t$

shows $\text{fterm-of-hterm } (\text{hterm-of-fterm } t) = t$

<proof>

lemma *fterms-of-hterms-hterms-of-fterms[simp]*:

assumes $ground_{ts} ts$
shows $fterms-of-hterms (hterms-of-fterms ts) = ts$
 $\langle proof \rangle$

lemma $ground-fterm-of-hterm$: $ground_t (fterm-of-hterm t)$
 $\langle proof \rangle$

lemma $ground-fterms-of-hterms$: $ground_{ts} (fterms-of-hterms ts)$
 $\langle proof \rangle$

1.3.2 Conversions - Literals and Herbrand Literals

fun $flit-of-hlit :: hterm literal \Rightarrow fterm literal$ **where**
 $flit-of-hlit (Pos p ts) = Pos p (fterms-of-hterms ts)$
 $| flit-of-hlit (Neg p ts) = Neg p (fterms-of-hterms ts)$

fun $hlit-of-flit :: fterm literal \Rightarrow hterm literal$ **where**
 $hlit-of-flit (Pos p ts) = Pos p (hterms-of-fterms ts)$
 $| hlit-of-flit (Neg p ts) = Neg p (hterms-of-fterms ts)$

lemma $ground-flit-of-hlit$: $ground_l (flit-of-hlit l)$
 $\langle proof \rangle$

theorem $hlit-of-flit-flit-of-hlit$ [simp]: $hlit-of-flit (flit-of-hlit l) = l$ $\langle proof \rangle$

theorem $flit-of-hlit-hlit-of-flit$ [simp]:
assumes $ground_l l$
shows $flit-of-hlit (hlit-of-flit l) = l$
 $\langle proof \rangle$

lemma $sign-flit-of-hlit$: $sign (flit-of-hlit l) = sign l$ $\langle proof \rangle$

lemma $hlit-of-flit-bij$: $bij\text{-betw } hlit\text{-of-flit } \{l. ground_l l\} UNIV$
 $\langle proof \rangle$

lemma $flit-of-hlit-bij$: $bij\text{-betw } flit\text{-of-hlit } UNIV \{l. ground_l l\}$
 $\langle proof \rangle$

1.3.3 Conversions - Atoms and Herbrand Atoms

fun $fatom-of-hatom :: hterm atom \Rightarrow fterm atom$ **where**
 $fatom-of-hatom (p, ts) = (p, fterms-of-hterms ts)$

fun $hatom-of-fatom :: fterm atom \Rightarrow hterm atom$ **where**
 $hatom-of-fatom (p, ts) = (p, hterms-of-fterms ts)$

lemma $ground-fatom-of-hatom$: $ground_{ts} (snd (fatom-of-hatom a))$
 $\langle proof \rangle$

theorem *hatom-of-fatome-fatome-of-hatome* [simp]: *hatome-of-fatome (fatome-of-hatome l) = l*
⟨proof⟩

theorem *fatome-of-hatome-hatome-of-fatome* [simp]:
assumes *ground_{ts} (snd l)*
shows *fatome-of-hatome (hatome-of-fatome l) = l*
⟨proof⟩

lemma *hatome-of-fatome-bij*: *bij-betw hatome-of-fatome ground-fatoms UNIV*
⟨proof⟩

lemma *fatome-of-hatome-bij*: *bij-betw fatome-of-hatome UNIV ground-fatoms*
⟨proof⟩

1.4 Enumerations

1.4.1 Enumerating Strings

definition *nat-of-string*:: *string ⇒ nat* **where**
nat-of-string ≡ (SOME f. bij f)

definition *string-of-nat*:: *nat ⇒ string* **where**
string-of-nat ≡ inv nat-of-string

lemma *nat-of-string-bij*: *bij nat-of-string*
⟨proof⟩

lemma *string-of-nat-bij*: *bij string-of-nat* ⟨proof⟩

lemma *nat-of-string-string-of-nat*[simp]: *nat-of-string (string-of-nat n) = n*
⟨proof⟩

lemma *string-of-nat-nat-of-string*[simp]: *string-of-nat (nat-of-string n) = n*
⟨proof⟩

1.4.2 Enumerating Herbrand Atoms

definition *nat-of-hatome*:: *hterm atom ⇒ nat* **where**
nat-of-hatome ≡ (SOME f. bij f)

definition *hatome-of-nat*:: *nat ⇒ hterm atom* **where**
hatome-of-nat ≡ inv nat-of-hatome

instantiation *hterm* :: *countable* **begin**

instance ⟨proof⟩

end

lemma *infinite-hatoms*: *infinite (UNIV :: ('t atom) set)*
⟨proof⟩

lemma *nat-of-hatom-bij*: *bij nat-of-hatom*
⟨*proof*⟩

lemma *hatom-of-nat-bij*: *bij hatom-of-nat* ⟨*proof*⟩

lemma *nat-of-hatom-hatom-of-nat[simp]*: *nat-of-hatom (hatom-of-nat n) = n*
⟨*proof*⟩

lemma *hatom-of-nat-nat-of-hatom[simp]*: *hatom-of-nat (nat-of-hatom l) = l*
⟨*proof*⟩

1.4.3 Enumerating Ground Atoms

definition *fatom-of-nat* :: *nat* \Rightarrow *fterm atom* **where**
fatom-of-nat = (λn . *fatom-of-hatom (hatom-of-nat n)*)

definition *nat-of-fatom* :: *fterm atom* \Rightarrow *nat* **where**
nat-of-fatom = (λt . *nat-of-hatom (hatom-of-fatom t)*)

theorem *diag-undia-fatom[simp]*:
assumes *ground_{ts} ts*
shows *fatom-of-nat (nat-of-fatom (p,ts)) = (p,ts)*
⟨*proof*⟩

theorem *undia-dia-fatom[simp]*: *nat-of-fatom (fatom-of-nat n) = n* ⟨*proof*⟩

lemma *fatom-of-nat-bij*: *bij-betw fatom-of-nat UNIV ground-fatoms*
⟨*proof*⟩

lemma *ground-fatom-of-nat*: *ground_{ts} (snd (fatom-of-nat x))* ⟨*proof*⟩

lemma *nat-of-fatom-bij*: *bij-betw nat-of-fatom ground-fatoms UNIV*
⟨*proof*⟩

end

2 Trees

theory *Tree* **imports** *Main* **begin**

Sometimes it is nice to think of *bools* as directions in a binary tree

hide-const (**open**) *Left Right*

type-synonym *dir* = *bool*

definition *Left* :: *bool* **where** *Left* = *True*

definition *Right* :: *bool* **where** *Right* = *False*

declare *Left-def* [*simp*]

declare *Right-def* [*simp*]

```

datatype tree =
  Leaf
| Branching (ltree: tree) (rtree: tree)

```

2.1 Sizes

```

fun treesize :: tree  $\Rightarrow$  nat where
  treesize Leaf = 0
| treesize (Branching l r) = 1 + treesize l + treesize r

```

```

lemma treesize-Leaf:
  assumes treesize T = 0
  shows T = Leaf
  <proof>

```

```

lemma treesize-Branching:
  assumes treesize T = Suc n
  shows  $\exists$  l r. T = Branching l r
  <proof>

```

2.2 Paths

```

fun path :: dir list  $\Rightarrow$  tree  $\Rightarrow$  bool where
  path [] T  $\longleftrightarrow$  True
| path (d#ds) (Branching T1 T2)  $\longleftrightarrow$  (if d then path ds T1 else path ds T2)
| path - -  $\longleftrightarrow$  False

```

```

lemma path-inv-Leaf: path p Leaf  $\longleftrightarrow$  p = []
  <proof>

```

```

lemma path-inv-Cons: path (a#ds) T  $\longrightarrow$  ( $\exists$  l r. T=Branching l r)
  <proof>

```

```

lemma path-inv-Branching-Left: path (Left#p) (Branching l r)  $\longleftrightarrow$  path p l
  <proof>

```

```

lemma path-inv-Branching-Right: path (Right#p) (Branching l r)  $\longleftrightarrow$  path p r
  <proof>

```

```

lemma path-inv-Branching:
  path p (Branching l r)  $\longleftrightarrow$  (p=[]  $\vee$  ( $\exists$  a p'. p=a#p'  $\wedge$  (a  $\longrightarrow$  path p' l)  $\wedge$  ( $\neg$ a
 $\longrightarrow$  path p' r))) (is ?L  $\longleftrightarrow$  ?R)
  <proof>

```

```

lemma path-prefix:
  assumes path (ds1@ds2) T
  shows path ds1 T
  <proof>

```


2.3 Branches

fun *branch* :: *dir list* \Rightarrow *tree* \Rightarrow *bool* **where**

branch [] *Leaf* \longleftrightarrow *True*
 | *branch* (*d* # *ds*) (*Branching l r*) \longleftrightarrow (*if d then branch ds l else branch ds r*)
 | *branch* - - \longleftrightarrow *False*

lemma *has-branch*: $\exists b. \text{branch } b \ T$

<proof>

lemma *branch-inv-Leaf*: *branch b Leaf* $\longleftrightarrow b = []$

<proof>

lemma *branch-inv-Branching-Left*:

branch (Left#b) (Branching l r) \longleftrightarrow *branch b l*
<proof>

lemma *branch-inv-Branching-Right*:

branch (Right#b) (Branching l r) \longleftrightarrow *branch b r*
<proof>

lemma *branch-inv-Branching*:

branch b (Branching l r) \longleftrightarrow
 ($\exists a \ b'. \ b = a \# \ b' \wedge (a \longrightarrow \text{branch } b' \ l) \wedge (\neg a \longrightarrow \text{branch } b' \ r)$)
<proof>

lemma *branch-inv-Leaf2*:

$T = \text{Leaf} \longleftrightarrow (\forall b. \text{branch } b \ T \longrightarrow b = [])$
<proof>

lemma *branch-is-path*:

assumes *branch ds T*
shows *path ds T*

<proof>

lemma *Branching-Leaf-Leaf-Tree*:

assumes $T = \text{Branching } T1 \ T2$
shows ($\exists B. \text{branch } (B@[True]) \ T \wedge \text{branch } (B@[False]) \ T$)

<proof>

2.4 Internal Paths

fun *internal* :: *dir list* \Rightarrow *tree* \Rightarrow *bool* **where**

internal [] (*Branching l r*) \longleftrightarrow *True*
 | *internal* (*d*#*ds*) (*Branching l r*) \longleftrightarrow (*if d then internal ds l else internal ds r*)
 | *internal* - - \longleftrightarrow *False*

lemma *internal-inv-Leaf*: $\neg \text{internal } b \ \text{Leaf}$ *<proof>*

lemma *internal-inv-Branching-Left*:

$internal (Left\#b) (Branching\ l\ r) \longleftrightarrow internal\ b\ l$ $\langle proof \rangle$

lemma *internal-inv-Branching-Right:*

$internal (Right\#b) (Branching\ l\ r) \longleftrightarrow internal\ b\ r$
 $\langle proof \rangle$

lemma *internal-inv-Branching:*

$internal\ p (Branching\ l\ r) \longleftrightarrow (p = [] \vee (\exists a\ p'. p = a\#p' \wedge (a \longrightarrow internal\ p'\ l) \wedge (\neg a \longrightarrow internal\ p'\ r)))$ (**is** $?L \longleftrightarrow ?R$)
 $\langle proof \rangle$

lemma *internal-is-path:*

assumes $internal\ ds\ T$
shows $path\ ds\ T$
 $\langle proof \rangle$

lemma *internal-prefix:*

assumes $internal (ds1@ds2@[d])\ T$
shows $internal\ ds1\ T$
 $\langle proof \rangle$

lemma *internal-branch:*

assumes $branch (ds1@ds2@[d])\ T$
shows $internal\ ds1\ T$
 $\langle proof \rangle$

fun $parent :: dir\ list \Rightarrow dir\ list$ **where**

$parent\ ds = tl\ ds$

2.5 Deleting Nodes

fun $delete :: dir\ list \Rightarrow tree \Rightarrow tree$ **where**

$delete\ []\ T = Leaf$
 $| delete (True\#ds) (Branching\ T_1\ T_2) = Branching (delete\ ds\ T_1)\ T_2$
 $| delete (False\#ds) (Branching\ T_1\ T_2) = Branching\ T_1 (delete\ ds\ T_2)$
 $| delete (a\#ds) Leaf = Leaf$

lemma *delete-Leaf:* $delete\ T\ Leaf = Leaf$ $\langle proof \rangle$

lemma *path-delete:*

assumes $path\ p (delete\ ds\ T)$
shows $path\ p\ T$
 $\langle proof \rangle$

lemma *branch-delete:*

assumes $branch\ p (delete\ ds\ T)$
shows $branch\ p\ T \vee p = ds$

<proof>

lemma *branch-delete-postfix*:
 assumes *path p (delete ds T)*
 shows $\neg(\exists c cs. p = ds @ c\#cs)$
<proof>

lemma *treezise-delete*:
 assumes *internal p T*
 shows *treezise (delete p T) < treezise T*
<proof>

fun *cutoff* :: (*dir list* \Rightarrow *bool*) \Rightarrow *dir list* \Rightarrow *tree* \Rightarrow *tree* **where**
 cutoff red ds (Branching T₁ T₂) =
 (*if red ds then Leaf else Branching (cutoff red (ds@[Left]) T₁) (cutoff red*
 (*ds@[Right]) T₂)*)
 | *cutoff red ds Leaf = Leaf*

Initially you should call *cutoff* with *ds* = []. If all branches are red, then *cutoff* gives a subtree. If all branches are red, then so are the ones in *cutoff*. The internal paths of *cutoff* are not red.

lemma *treezise-cutoff*: *treezise (cutoff red ds T) \leq treezise T*
<proof>

abbreviation *anypath* :: *tree* \Rightarrow (*dir list* \Rightarrow *bool*) \Rightarrow *bool* **where**
 anypath T P \equiv $\forall p. path p T \longrightarrow P p$

abbreviation *anybranch* :: *tree* \Rightarrow (*dir list* \Rightarrow *bool*) \Rightarrow *bool* **where**
 anybranch T P \equiv $\forall p. branch p T \longrightarrow P p$

abbreviation *anyinternal* :: *tree* \Rightarrow (*dir list* \Rightarrow *bool*) \Rightarrow *bool* **where**
 anyinternal T P \equiv $\forall p. internal p T \longrightarrow P p$

lemma *cutoff-branch'*:
 assumes *anybranch T ($\lambda b. red(ds@b)$)*
 shows *anybranch (cutoff red ds T) ($\lambda b. red(ds@b)$)*
<proof>

lemma *cutoff-branch*:
 assumes *anybranch T ($\lambda p. red p$)*
 shows *anybranch (cutoff red [] T) ($\lambda p. red p$)*
<proof>

lemma *cutoff-internal'*:
 assumes *anybranch T ($\lambda b. red(ds@b)$)*
 shows *anyinternal (cutoff red ds T) ($\lambda b. \neg red(ds@b)$)*
<proof>

lemma *cutoff-internal*:

assumes *anybranch* T *red*

shows *anyinternal* (*cutoff red* \square T) ($\lambda p. \neg red\ p$)

$\langle proof \rangle$

lemma *cutoff-branch-internal'*:

assumes *anybranch* T *red*

shows *anyinternal* (*cutoff red* \square T) ($\lambda p. \neg red\ p$) \wedge *anybranch* (*cutoff red* \square T)
($\lambda p. red\ p$)

$\langle proof \rangle$

lemma *cutoff-branch-internal*:

assumes *anybranch* T *red*

shows $\exists T'. anyinternal\ T'$ ($\lambda p. \neg red\ p$) \wedge *anybranch* T' ($\lambda p. red\ p$)

$\langle proof \rangle$

3 Possibly Infinite Trees

Possibly infinite trees are of type *dir list set*.

abbreviation *wf-tree* $:: dir\ list\ set \Rightarrow bool$ **where**

wf-tree $T \equiv (\forall ds\ d. (ds\ @\ d) \in T \longrightarrow ds \in T)$

The subtree in with root r

fun *subtree* $:: dir\ list\ set \Rightarrow dir\ list \Rightarrow dir\ list\ set$ **where**

subtree $T\ r = \{ds \in T. \exists ds'. ds = r\ @\ ds'\}$

A subtree of a tree is either in the left branch, the right branch, or is the tree itself

lemma *subtree-pos*:

subtree $T\ ds \subseteq subtree\ T\ (ds\ @\ [Left]) \cup subtree\ T\ (ds\ @\ [Right]) \cup \{ds\}$
 $\langle proof \rangle$

3.1 Infinite Paths

abbreviation *wf-infpath* $:: (nat \Rightarrow 'a\ list) \Rightarrow bool$ **where**

wf-infpath $f \equiv (f\ 0 = []) \wedge (\forall n. \exists a. f\ (Suc\ n) = (f\ n)\ @\ [a])$

lemma *infpath-length*:

assumes *wf-infpath* f

shows *length* ($f\ n$) = n

$\langle proof \rangle$

lemma *chain-prefix*:

assumes *wf-infpath* f

assumes $n_1 \leq n_2$

shows $\exists a. (f\ n_1)\ @\ a = (f\ n_2)$

$\langle proof \rangle$

If we make a lookup in a list, then looking up in an extension gives us the same value.

lemma *ith-in-extension*:
assumes *chain*: *wf-infnpath* *f*
assumes *smalli*: $i < \text{length } (f \ n_1)$
assumes $n_1 \ n_2$: $n_1 \leq n_2$
shows $f \ n_1 \ ! \ i = f \ n_2 \ ! \ i$
 $\langle \text{proof} \rangle$

4 König's Lemma

lemma *inf-subs*:
assumes *inf*: $\neg \text{finite}(\text{subtree } T \ ds)$
shows $\neg \text{finite}(\text{subtree } T \ (ds \ @ \ [\text{Left}])) \vee \neg \text{finite}(\text{subtree } T \ (ds \ @ \ [\text{Right}]))$
 $\langle \text{proof} \rangle$

fun *buildchain* :: $(\text{dir list} \Rightarrow \text{dir list}) \Rightarrow \text{nat} \Rightarrow \text{dir list}$ **where**
buildchain *next* 0 = []
| *buildchain* *next* (Suc *n*) = *next* (*buildchain* *next* *n*)

lemma *konig*:
assumes *inf*: $\neg \text{finite } T$
assumes *wellformed*: *wf-tree* *T*
shows $\exists c. \text{wf-infnpath } c \wedge (\forall n. (c \ n) \in T)$
 $\langle \text{proof} \rangle$

end

5 More Terms and Literals

theory *Resolution* **imports** *TermsAndLiterals* *Tree* **begin**

fun *complement* :: $'t \ \text{literal} \Rightarrow 't \ \text{literal}$ ($-\ ^c$ [300] 300) **where**
 $(\text{Pos } P \ ts)^c = \text{Neg } P \ ts$
| $(\text{Neg } P \ ts)^c = \text{Pos } P \ ts$

lemma *cancel-comp1*: $(l^c)^c = l$ $\langle \text{proof} \rangle$

lemma *cancel-comp2*:
assumes *asm*: $l_1^c = l_2^c$
shows $l_1 = l_2$
 $\langle \text{proof} \rangle$

lemma *comp-exi1*: $\exists l'. l' = l^c$ $\langle \text{proof} \rangle$

lemma *comp-exi2*: $\exists l. l' = l^c$
 $\langle \text{proof} \rangle$

lemma *comp-swap*: $l_1^c = l_2 \longleftrightarrow l_1 = l_2^c$
 ⟨proof⟩

lemma *sign-comp*: $sign\ l_1 \neq sign\ l_2 \wedge get\text{-pred}\ l_1 = get\text{-pred}\ l_2 \wedge get\text{-terms}\ l_1 = get\text{-terms}\ l_2 \longleftrightarrow l_2 = l_1^c$
 ⟨proof⟩

lemma *sign-comp-atom*: $sign\ l_1 \neq sign\ l_2 \wedge get\text{-atom}\ l_1 = get\text{-atom}\ l_2 \longleftrightarrow l_2 = l_1^c$
 ⟨proof⟩

6 Clauses

type-synonym *'t clause* = *'t literal set*

abbreviation *complement*_s :: *'t literal set* \Rightarrow *'t literal set* $(-^C [300] 300)$ **where**
 $L^C \equiv complement\ 'L$

lemma *cancel-compl1*: $(L^C)^C = L$
 ⟨proof⟩

lemma *cancel-compl2*:
assumes *asm*: $L_1^C = L_2^C$
shows $L_1 = L_2$
 ⟨proof⟩

fun *vars_t* :: *fterm* \Rightarrow *var-sym set* **where**
 $vars_t\ (Var\ x) = \{x\}$
 $| vars_t\ (Fun\ f\ ts) = (\bigcup t \in set\ ts.\ vars_t\ t)$

abbreviation *vars_{ts}* :: *fterm list* \Rightarrow *var-sym set* **where**
 $vars_{ts}\ ts \equiv (\bigcup t \in set\ ts.\ vars_t\ t)$

definition *vars_l* :: *fterm literal* \Rightarrow *var-sym set* **where**
 $vars_l\ l = vars_{ts}\ (get\text{-terms}\ l)$

definition *vars_{ls}* :: *fterm literal set* \Rightarrow *var-sym set* **where**
 $vars_{ls}\ L \equiv \bigcup l \in L.\ vars_l\ l$

lemma *ground-vars_t*:
assumes *ground_t* *t*
shows $vars_t\ t = \{\}$
 ⟨proof⟩

lemma *ground_{ts}-vars_{ts}*:
assumes *ground_{ts}* *ts*
shows $vars_{ts}\ ts = \{\}$
 ⟨proof⟩

lemma *ground_l-vars_l*:
assumes *ground_l l*
shows *vars_l l = {}*
 \langle *proof* \rangle

lemma *ground_{l_s}-vars_{l_s}*:
assumes *ground_{l_s} L*
shows *vars_{l_s} L = {}* \langle *proof* \rangle

lemma *ground-comp*: *ground_l (l^c) \longleftrightarrow ground_l l* \langle *proof* \rangle

lemma *ground-compl_s*: *ground_{l_s} (L^C) \longleftrightarrow ground_{l_s} L* \langle *proof* \rangle

7 Semantics

type-synonym *'u fun-denot = fun-sym \Rightarrow 'u list \Rightarrow 'u*

type-synonym *'u pred-denot = pred-sym \Rightarrow 'u list \Rightarrow bool*

type-synonym *'u var-denot = var-sym \Rightarrow 'u*

fun *eval_t :: 'u var-denot \Rightarrow 'u fun-denot \Rightarrow fterm \Rightarrow 'u where*
eval_t E F (Var x) = E x
 $|$ *eval_t E F (Fun f ts) = F f (map (eval_t E F) ts)*

abbreviation *eval_{ts} :: 'u var-denot \Rightarrow 'u fun-denot \Rightarrow fterm list \Rightarrow 'u list where*
eval_{ts} E F ts \equiv map (eval_t E F) ts

fun *eval_l :: 'u var-denot \Rightarrow 'u fun-denot \Rightarrow 'u pred-denot \Rightarrow fterm literal \Rightarrow bool*
where
eval_l E F G (Pos p ts) \longleftrightarrow G p (eval_{ts} E F ts)
 $|$ *eval_l E F G (Neg p ts) \longleftrightarrow \neg G p (eval_{ts} E F ts)*

definition *eval_c :: 'u fun-denot \Rightarrow 'u pred-denot \Rightarrow fterm clause \Rightarrow bool where*
eval_c F G C \longleftrightarrow (\forall E. \exists l \in C. eval_l E F G l)

definition *eval_{cs} :: 'u fun-denot \Rightarrow 'u pred-denot \Rightarrow fterm clause set \Rightarrow bool where*
eval_{cs} F G Cs \longleftrightarrow (\forall C \in Cs. eval_c F G C)

7.1 Semantics of Ground Terms

lemma *ground-var-denott*:
assumes *ground_t t*
shows *eval_t E F t = eval_t E' F t*
 \langle *proof* \rangle

lemma *ground-var-denotts*:
assumes *ground_{ts} ts*
shows *eval_{ts} E F ts = eval_{ts} E' F ts*
 \langle *proof* \rangle

lemma *ground-var-denot*:
assumes *ground_l* *l*
shows $eval_l E F G l = eval_l E' F G l$
 $\langle proof \rangle$

8 Substitutions

type-synonym *substitution* = *var-sym* \Rightarrow *fterm*

fun *sub* :: *fterm* \Rightarrow *substitution* \Rightarrow *fterm* (**infixl** \cdot_t 55) **where**
 $(Var\ x) \cdot_t \sigma = \sigma\ x$
 $| (Fun\ f\ ts) \cdot_t \sigma = Fun\ f\ (map\ (\lambda t. t \cdot_t \sigma)\ ts)$

abbreviation *subs* :: *fterm list* \Rightarrow *substitution* \Rightarrow *fterm list* (**infixl** \cdot_{ts} 55) **where**
 $ts \cdot_{ts} \sigma \equiv (map\ (\lambda t. t \cdot_t \sigma)\ ts)$

fun *subl* :: *fterm literal* \Rightarrow *substitution* \Rightarrow *fterm literal* (**infixl** \cdot_l 55) **where**
 $(Pos\ p\ ts) \cdot_l \sigma = Pos\ p\ (ts \cdot_{ts} \sigma)$
 $| (Neg\ p\ ts) \cdot_l \sigma = Neg\ p\ (ts \cdot_{ts} \sigma)$

abbreviation *subls* :: *fterm literal set* \Rightarrow *substitution* \Rightarrow *fterm literal set* (**infixl** \cdot_{ls} 55) **where**
 $L \cdot_{ls} \sigma \equiv (\lambda l. l \cdot_l \sigma) \cdot L$

lemma *subls-def2*: $L \cdot_{ls} \sigma = \{l \cdot_l \sigma \mid l. l \in L\}$ $\langle proof \rangle$

definition *instance-of_t* :: *fterm* \Rightarrow *fterm* \Rightarrow *bool* **where**
 $instance-of_t\ t_1\ t_2 \longleftrightarrow (\exists \sigma. t_1 = t_2 \cdot_t \sigma)$

definition *instance-of_{ts}* :: *fterm list* \Rightarrow *fterm list* \Rightarrow *bool* **where**
 $instance-of_{ts}\ ts_1\ ts_2 \longleftrightarrow (\exists \sigma. ts_1 = ts_2 \cdot_{ts} \sigma)$

definition *instance-of_l* :: *fterm literal* \Rightarrow *fterm literal* \Rightarrow *bool* **where**
 $instance-of_l\ l_1\ l_2 \longleftrightarrow (\exists \sigma. l_1 = l_2 \cdot_l \sigma)$

definition *instance-of_{ls}* :: *fterm clause* \Rightarrow *fterm clause* \Rightarrow *bool* **where**
 $instance-of_{ls}\ C_1\ C_2 \longleftrightarrow (\exists \sigma. C_1 = C_2 \cdot_{ls} \sigma)$

lemma *comp-sub*: $(l^c) \cdot_l \sigma = (l \cdot_l \sigma)^c$
 $\langle proof \rangle$

lemma *compls-subls*: $(L^C) \cdot_{ls} \sigma = (L \cdot_{ls} \sigma)^C$
 $\langle proof \rangle$

lemma *subls-union*: $(L_1 \cup L_2) \cdot_{ls} \sigma = (L_1 \cdot_{ls} \sigma) \cup (L_2 \cdot_{ls} \sigma)$ $\langle proof \rangle$

definition *var-renaming-of* :: *fterm clause* \Rightarrow *fterm clause* \Rightarrow *bool* **where**

$var-renaming-of\ C_1\ C_2 \longleftrightarrow instance-of_{I_s}\ C_1\ C_2 \wedge instance-of_{I_s}\ C_2\ C_1$

8.1 The Empty Substitution

abbreviation $\varepsilon :: substitution\ where$

$\varepsilon \equiv Var$

lemma *empty-subst*: $(t :: fterm) \cdot_t \varepsilon = t$
<proof>

lemma *empty-substs*: $ts \cdot_{ts} \varepsilon = ts$
<proof>

lemma *empty-subl*: $l \cdot_l \varepsilon = l$
<proof>

lemma *empty-subls*: $L \cdot_{I_s} \varepsilon = L$
<proof>

lemma *instance-of_t-self*: $instance-of_t\ t\ t$
<proof>

lemma *instance-of_{ts}-self*: $instance-of_{ts}\ ts\ ts$
<proof>

lemma *instance-of_l-self*: $instance-of_l\ l\ l$
<proof>

lemma *instance-of_{I_s}-self*: $instance-of_{I_s}\ L\ L$
<proof>

8.2 Substitutions and Ground Terms

lemma *ground-sub*:
 assumes $ground_t\ t$
 shows $t \cdot_t \sigma = t$
<proof>

lemma *ground-subs*:
 assumes $ground_{ts}\ ts$
 shows $ts \cdot_{ts} \sigma = ts$
<proof>

lemma *ground_l-subs*:
 assumes $ground_l\ l$
 shows $l \cdot_l \sigma = l$
<proof>

lemma *ground_{I_s}-subls*:
 assumes $ground$: $ground_{I_s}\ L$

shows $L \cdot_{ls} \sigma = L$
<proof>

8.3 Composition

definition *composition* :: *substitution* \Rightarrow *substitution* \Rightarrow *substitution* (**infixl** · 55)
where

$$(\sigma_1 \cdot \sigma_2) x = (\sigma_1 x) \cdot_t \sigma_2$$

lemma *composition-conseq2t*: $(t \cdot_t \sigma_1) \cdot_t \sigma_2 = t \cdot_t (\sigma_1 \cdot \sigma_2)$
<proof>

lemma *composition-conseq2ts*: $(ts \cdot_{ts} \sigma_1) \cdot_{ts} \sigma_2 = ts \cdot_{ts} (\sigma_1 \cdot \sigma_2)$
<proof>

lemma *composition-conseq2l*: $(l \cdot_l \sigma_1) \cdot_l \sigma_2 = l \cdot_l (\sigma_1 \cdot \sigma_2)$
<proof>

lemma *composition-conseq2ls*: $(L \cdot_{ls} \sigma_1) \cdot_{ls} \sigma_2 = L \cdot_{ls} (\sigma_1 \cdot \sigma_2)$
<proof>

lemma *composition-assoc*: $\sigma_1 \cdot (\sigma_2 \cdot \sigma_3) = (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$
<proof>

lemma *empty-comp1*: $(\sigma \cdot \varepsilon) = \sigma$
<proof>

lemma *empty-comp2*: $(\varepsilon \cdot \sigma) = \sigma$
<proof>

lemma *instance-of_t-trans* :
 assumes t_{12} : *instance-of_t* $t_1 t_2$
 assumes t_{23} : *instance-of_t* $t_2 t_3$
 shows *instance-of_t* $t_1 t_3$
<proof>

lemma *instance-of_{ts}-trans* :
 assumes ts_{12} : *instance-of_{ts}* $ts_1 ts_2$
 assumes ts_{23} : *instance-of_{ts}* $ts_2 ts_3$
 shows *instance-of_{ts}* $ts_1 ts_3$
<proof>

lemma *instance-of_l-trans* :
 assumes l_{12} : *instance-of_l* $l_1 l_2$
 assumes l_{23} : *instance-of_l* $l_2 l_3$
 shows *instance-of_l* $l_1 l_3$
<proof>

lemma *instance-of_{l_s}-trans* :
assumes L_{12} : *instance-of_{l_s}* L_1 L_2
assumes L_{23} : *instance-of_{l_s}* L_2 L_3
shows *instance-of_{l_s}* L_1 L_3
⟨*proof*⟩

8.4 Merging substitutions

lemma *project-sub*:
assumes *inst-C*: $C \cdot_{l_s} \text{lmbd} = C'$
assumes *L'sub*: $L' \subseteq C'$
shows $\exists L \subseteq C. L \cdot_{l_s} \text{lmbd} = L' \wedge (C-L) \cdot_{l_s} \text{lmbd} = C' - L'$
⟨*proof*⟩

lemma *relevant-vars-subt*:
assumes $\forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x$
shows $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
⟨*proof*⟩

lemma *relevant-vars-subts*:
assumes *asm*: $\forall x \in \text{vars}_{t_s} t_s. \sigma_1 x = \sigma_2 x$
shows $t_s \cdot_{t_s} \sigma_1 = t_s \cdot_{t_s} \sigma_2$
⟨*proof*⟩

lemma *relevant-vars-subl*:
assumes $\forall x \in \text{vars}_l l. \sigma_1 x = \sigma_2 x$
shows $l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
⟨*proof*⟩

lemma *relevant-vars-subls*:
assumes *asm*: $\forall x \in \text{vars}_{l_s} L. \sigma_1 x = \sigma_2 x$
shows $L \cdot_{l_s} \sigma_1 = L \cdot_{l_s} \sigma_2$
⟨*proof*⟩

lemma *merge-sub*:
assumes *dist*: $\text{vars}_{l_s} C \cap \text{vars}_{l_s} D = \{\}$
assumes *CC'*: $C \cdot_{l_s} \text{lmbd} = C'$
assumes *DD'*: $D \cdot_{l_s} \mu = D'$
shows $\exists \eta. C \cdot_{l_s} \eta = C' \wedge D \cdot_{l_s} \eta = D'$
⟨*proof*⟩

8.5 Standardizing apart

abbreviation *std₁* :: *fterm clause* \Rightarrow *fterm clause* **where**
 $\text{std}_1 C \equiv C \cdot_{l_s} (\lambda x. \text{Var} ("1" @ x))$

abbreviation *std₂* :: *fterm clause* \Rightarrow *fterm clause* **where**
 $\text{std}_2 C \equiv C \cdot_{l_s} (\lambda x. \text{Var} ("2" @ x))$

lemma *std-apart-apart''*:

assumes $x \in \text{vars}_t$ ($t \cdot_t (\lambda x::\text{char list. Var } (y @ x))$)
shows $\exists x'. x = y@x'$
 $\langle \text{proof} \rangle$

lemma *std-apart-apart'*:
assumes $x \in \text{vars}_l$ ($l \cdot_l (\lambda x. \text{Var } (y@x))$)
shows $\exists x'. x = y@x'$
 $\langle \text{proof} \rangle$

lemma *std-apart-apart*: $\text{vars}_{l_s} (\text{std}_1 C_1) \cap \text{vars}_{l_s} (\text{std}_2 C_2) = \{\}$
 $\langle \text{proof} \rangle$

lemma *std-apart-instance-of_{l_s}1*: $\text{instance-of}_{l_s} C_1 (\text{std}_1 C_1)$
 $\langle \text{proof} \rangle$

lemma *std-apart-instance-of_{l_s}2*: $\text{instance-of}_{l_s} C_2 (\text{std}_2 C_2)$
 $\langle \text{proof} \rangle$

9 Unifiers

definition $\text{unifier}_{ts} :: \text{substitution} \Rightarrow \text{fterm set} \Rightarrow \text{bool}$ **where**
 $\text{unifier}_{ts} \sigma ts \longleftrightarrow (\exists t'. \forall t \in ts. t \cdot_t \sigma = t')$

definition $\text{unifier}_{l_s} :: \text{substitution} \Rightarrow \text{fterm literal set} \Rightarrow \text{bool}$ **where**
 $\text{unifier}_{l_s} \sigma L \longleftrightarrow (\exists l'. \forall l \in L. l \cdot_l \sigma = l')$

lemma *unif-sub*:
assumes *unif*: $\text{unifier}_{l_s} \sigma L$
assumes *nonempty*: $L \neq \{\}$
shows $\exists l. \text{subls } L \sigma = \{\text{subl } l \sigma\}$
 $\langle \text{proof} \rangle$

lemma *unifiert-def2*:
assumes *L-elem*: $ts \neq \{\}$
shows $\text{unifier}_{ts} \sigma ts \longleftrightarrow (\exists l. (\lambda t. \text{sub } t \sigma) ' ts = \{l\})$
 $\langle \text{proof} \rangle$

lemma *unifier_{l_s}-def2*:
assumes *L-elem*: $L \neq \{\}$
shows $\text{unifier}_{l_s} \sigma L \longleftrightarrow (\exists l. L \cdot_{l_s} \sigma = \{l\})$
 $\langle \text{proof} \rangle$

lemma *ground_{l_s}-unif-singleton*:
assumes *ground_{l_s}*: $\text{ground}_{l_s} L$
assumes *unif*: $\text{unifier}_{l_s} \sigma' L$
assumes *empt*: $L \neq \{\}$
shows $\exists l. L = \{l\}$
 $\langle \text{proof} \rangle$

definition *unifiablets* :: *fterm set* \Rightarrow *bool* **where**
unifiablets *fs* $\longleftrightarrow (\exists \sigma. \text{unifier}_{ts} \sigma \text{ fs})$

definition *unifiablels* :: *fterm literal set* \Rightarrow *bool* **where**
unifiablels *L* $\longleftrightarrow (\exists \sigma. \text{unifier}_{ls} \sigma L)$

lemma *unifier-comp[simp]*: $\text{unifier}_{ls} \sigma (L^C) \longleftrightarrow \text{unifier}_{ls} \sigma L$
 $\langle \text{proof} \rangle$

lemma *unifier-sub1*:
assumes $\text{unifier}_{ls} \sigma L$
assumes $L' \subseteq L$
shows $\text{unifier}_{ls} \sigma L'$
 $\langle \text{proof} \rangle$

lemma *unifier-sub2*:
assumes *asm*: $\text{unifier}_{ls} \sigma (L_1 \cup L_2)$
shows $\text{unifier}_{ls} \sigma L_1 \wedge \text{unifier}_{ls} \sigma L_2$
 $\langle \text{proof} \rangle$

9.1 Most General Unifiers

definition *mgu_{ts}* :: *substitution* \Rightarrow *fterm set* \Rightarrow *bool* **where**
mgu_{ts} σ *ts* $\longleftrightarrow \text{unifier}_{ts} \sigma \text{ ts} \wedge (\forall u. \text{unifier}_{ts} u \text{ ts} \longrightarrow (\exists i. u = \sigma \cdot i))$

definition *mgu_{ls}* :: *substitution* \Rightarrow *fterm literal set* \Rightarrow *bool* **where**
mgu_{ls} σ *L* $\longleftrightarrow \text{unifier}_{ls} \sigma L \wedge (\forall u. \text{unifier}_{ls} u L \longrightarrow (\exists i. u = \sigma \cdot i))$

10 Resolution

definition *applicable* :: *fterm clause* \Rightarrow *fterm clause*
 \Rightarrow *fterm literal set* \Rightarrow *fterm literal set*
 \Rightarrow *substitution* \Rightarrow *bool* **where**

applicable $C_1 C_2 L_1 L_2 \sigma \longleftrightarrow$
 $C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
 $\wedge \text{vars}_{ls} C_1 \cap \text{vars}_{ls} C_2 = \{\}$
 $\wedge L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
 $\wedge \text{mgu}_{ls} \sigma (L_1 \cup L_2^C)$

definition *mresolution* :: *fterm clause* \Rightarrow *fterm clause*
 \Rightarrow *fterm literal set* \Rightarrow *fterm literal set*
 \Rightarrow *substitution* \Rightarrow *fterm clause* **where**
mresolution $C_1 C_2 L_1 L_2 \sigma = ((C_1 \cdot_{ls} \sigma) - (L_1 \cdot_{ls} \sigma)) \cup ((C_2 \cdot_{ls} \sigma) - (L_2 \cdot_{ls} \sigma))$
 $\langle \text{proof} \rangle$

definition *resolution* :: *fterm clause* \Rightarrow *fterm clause*
 \Rightarrow *fterm literal set* \Rightarrow *fterm literal set*
 \Rightarrow *substitution* \Rightarrow *fterm clause* **where**
resolution $C_1 C_2 L_1 L_2 \sigma = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot_{ls} \sigma$

inductive *mresolution-step* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

mresolution-rule:

$C_1 \in Cs \Rightarrow C_2 \in Cs \Rightarrow \text{applicable } C_1 C_2 L_1 L_2 \sigma \Rightarrow$
 $\text{mresolution-step } Cs (Cs \cup \{\text{mresolution } C_1 C_2 L_1 L_2 \sigma\})$

| *standardize-apart*:

$C \in Cs \Rightarrow \text{var-renaming-of } C C' \Rightarrow \text{mresolution-step } Cs (Cs \cup \{C'\})$

inductive *resolution-step* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

resolution-rule:

$C_1 \in Cs \Rightarrow C_2 \in Cs \Rightarrow \text{applicable } C_1 C_2 L_1 L_2 \sigma \Rightarrow$
 $\text{resolution-step } Cs (Cs \cup \{\text{resolution } C_1 C_2 L_1 L_2 \sigma\})$

| *standardize-apart*:

$C \in Cs \Rightarrow \text{var-renaming-of } C C' \Rightarrow \text{resolution-step } Cs (Cs \cup \{C'\})$

definition *mresolution-deriv* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

mresolution-deriv = *rtranclp mresolution-step*

definition *resolution-deriv* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

resolution-deriv = *rtranclp resolution-step*

11 Soundness

definition *evalsub* :: *'u var-denot* \Rightarrow *'u fun-denot* \Rightarrow *substitution* \Rightarrow *'u var-denot* **where**

evalsub *E F* σ = *eval_t* *E F* $\circ \sigma$

lemma *substitutiont*: *eval_t* *E F* (*t* \cdot_t σ) = *eval_t* (*evalsub* *E F* σ) *F t*

<proof>

lemma *substitutionts*: *eval_{ts}* *E F* (*ts* \cdot_{ts} σ) = *eval_{ts}* (*evalsub* *E F* σ) *F ts*

<proof>

lemma *substitutionl*: *eval_l* *E F G* (*l* \cdot_l σ) \longleftrightarrow *eval_l* (*evalsub* *E F* σ) *F G l*

<proof>

lemma *subst-sound*:

assumes *asm*: *eval_c* *F G C*

shows *eval_c* *F G* (*C* \cdot_{ls} σ)

<proof>

lemma *simple-resolution-sound*:

assumes *C₁sat*: *eval_c* *F G C₁*

assumes *C₂sat*: *eval_c* *F G C₂*

assumes *l₁inc₁*: *l₁ \in C₁*

assumes *l₂inc₂*: *l₂ \in C₂*

assumes *comp*: *l₁^c = l₂*

shows *eval_c* *F G* ($(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})$)

<proof>

lemma *mresolution-sound*:

assumes *sat*₁: $eval_c F G C_1$

assumes *sat*₂: $eval_c F G C_2$

assumes *appl*: *applicable* $C_1 C_2 L_1 L_2 \sigma$

shows $eval_c F G (mresolution C_1 C_2 L_1 L_2 \sigma)$

<proof>

lemma *resolution-superset*: $mresolution C_1 C_2 L_1 L_2 \sigma \subseteq resolution C_1 C_2 L_1$

$L_2 \sigma$

<proof>

lemma *superset-sound*:

assumes *sup*: $C \subseteq C'$

assumes *sat*: $eval_c F G C$

shows $eval_c F G C'$

<proof>

theorem *resolution-sound*:

assumes *sat*₁: $eval_c F G C_1$

assumes *sat*₂: $eval_c F G C_2$

assumes *appl*: *applicable* $C_1 C_2 L_1 L_2 \sigma$

shows $eval_c F G (resolution C_1 C_2 L_1 L_2 \sigma)$

<proof>

lemma *mstep-sound*:

assumes *mresolution-step* $Cs Cs'$

assumes $eval_{cs} F G Cs$

shows $eval_{cs} F G Cs'$

<proof>

theorem *step-sound*:

assumes *resolution-step* $Cs Cs'$

assumes $eval_{cs} F G Cs$

shows $eval_{cs} F G Cs'$

<proof>

lemma *mderivation-sound*:

assumes *mresolution-deriv* $Cs Cs'$

assumes $eval_{cs} F G Cs$

shows $eval_{cs} F G Cs'$

<proof>

theorem *derivation-sound*:

assumes *resolution-deriv* $Cs Cs'$

assumes $eval_{cs} F G Cs$

shows $eval_{cs} F G Cs'$

<proof>

theorem *derivation-sound-refute*:
assumes *deriv: resolution-deriv Cs Cs' \wedge $\{\}$ \in Cs'*
shows $\neg \text{eval}_{Cs} F G Cs$
 $\langle \text{proof} \rangle$

12 Herbrand Interpretations

HFun is the Herbrand function denotation in which terms are mapped to themselves.

term *HFun*

lemma *eval-ground_t*:
assumes *ground_t t*
shows $(\text{eval}_t E \text{HFun } t) = \text{hterm-of-fterm } t$
 $\langle \text{proof} \rangle$

lemma *eval-ground_{ts}*:
assumes *ground_{ts} ts*
shows $(\text{eval}_{ts} E \text{HFun } ts) = \text{hterms-of-fters } ts$
 $\langle \text{proof} \rangle$

lemma *eval_l-ground_{ts}*:
assumes *asm: ground_{ts} ts*
shows $\text{eval}_l E \text{HFun } G (\text{Pos } P ts) \longleftrightarrow G P (\text{hterms-of-fters } ts)$
 $\langle \text{proof} \rangle$

13 Partial Interpretations

type-synonym *partial-pred-denot = bool list*

definition *falsifies_l* :: *partial-pred-denot \Rightarrow fterm literal \Rightarrow bool **where**
 $\text{falsifies}_l G l \longleftrightarrow$
 $\text{ground}_l l$
 $\wedge (\text{let } i = \text{nat-of-fatom } (\text{get-atom } l) \text{ in}$
 $i < \text{length } G \wedge G ! i = (\neg \text{sign } l)$
 $)$*

A ground clause is falsified if it is actually ground and all its literals are falsified.

abbreviation *falsifies_g* :: *partial-pred-denot \Rightarrow fterm clause \Rightarrow bool **where**
 $\text{falsifies}_g G C \equiv \text{ground}_{ls} C \wedge (\forall l \in C. \text{falsifies}_l G l)$*

abbreviation *falsifies_c* :: *partial-pred-denot \Rightarrow fterm clause \Rightarrow bool **where**
 $\text{falsifies}_c G C \equiv (\exists C'. \text{instance-of}_{ls} C' C \wedge \text{falsifies}_g G C')$*

abbreviation *falsifies_{cs}* :: *partial-pred-denot \Rightarrow fterm clause set \Rightarrow bool **where***

$falsifies_{Cs} G Cs \equiv (\exists C \in Cs. falsifies_c G C)$

abbreviation $extend :: (nat \Rightarrow partial-pred-denot) \Rightarrow hterm\ pred-denot$ **where**
 $extend\ f\ P\ ts \equiv ($
 $\quad let\ n = nat-of-hatom\ (P, ts)\ in$
 $\quad\quad f\ (Suc\ n)\ !\ n$
 $\quad)$

fun $sub-of-denot :: hterm\ var-denot \Rightarrow substitution$ **where**
 $sub-of-denot\ E = fterm-of-hterm \circ E$

lemma $ground-sub-of-denott: ground_t\ (t \cdot_t\ (sub-of-denot\ E))$
 $\langle proof \rangle$

lemma $ground-sub-of-denotts: ground_{ts}\ (ts \cdot_{ts}\ sub-of-denot\ E)$
 $\langle proof \rangle$

lemma $ground-sub-of-denotl: ground_l\ (l \cdot_l\ sub-of-denot\ E)$
 $\langle proof \rangle$

lemma $sub-of-denot-equivx: eval_t\ E\ HFun\ (sub-of-denot\ E\ x) = E\ x$
 $\langle proof \rangle$

lemma $sub-of-denot-equivt:$
 $eval_t\ E\ HFun\ (t \cdot_t\ (sub-of-denot\ E)) = eval_t\ E\ HFun\ t$
 $\langle proof \rangle$

lemma $sub-of-denot-equivts: eval_{ts}\ E\ HFun\ (ts \cdot_{ts}\ (sub-of-denot\ E)) = eval_{ts}\ E$
 $HFun\ ts$
 $\langle proof \rangle$

lemma $sub-of-denot-equivl: eval_l\ E\ HFun\ G\ (l \cdot_l\ sub-of-denot\ E) \longleftrightarrow eval_l\ E$
 $HFun\ G\ l$
 $\langle proof \rangle$

Under an Herbrand interpretation, an environment is equivalent to a substitution.

lemma $sub-of-denot-equiv-ground'$:
 $eval_l\ E\ HFun\ G\ l = eval_l\ E\ HFun\ G\ (l \cdot_l\ sub-of-denot\ E) \wedge ground_l\ (l \cdot_l$
 $sub-of-denot\ E)$
 $\langle proof \rangle$

Under an Herbrand interpretation, an environment is similar to a substitution - also for partial interpretations.

lemma $partial-equiv-subst:$
assumes $falsifies_c\ G\ (C \cdot_{ts}\ \tau)$
shows $falsifies_c\ G\ C$

<proof>

Under an Herbrand interpretation, an environment is equivalent to a substitution.

lemma *sub-of-denot-equiv-ground*:

$((\exists l \in C. \text{eval}_l E \text{HFun } G l) \longleftrightarrow (\exists l \in C \cdot_{l_s} \text{sub-of-denot } E. \text{eval}_l E \text{HFun } G l))$

$\wedge \text{ground}_{l_s} (C \cdot_{l_s} \text{sub-of-denot } E)$

<proof>

lemma *std₁-falsifies*: $\text{falsifies}_c G C_1 \longleftrightarrow \text{falsifies}_c G (\text{std}_1 C_1)$

<proof>

lemma *std₂-falsifies*: $\text{falsifies}_c G C_2 \longleftrightarrow \text{falsifies}_c G (\text{std}_2 C_2)$

<proof>

lemma *std₁-renames*: *var-renaming-of* $C_1 (\text{std}_1 C_1)$

<proof>

lemma *std₂-renames*: *var-renaming-of* $C_2 (\text{std}_2 C_2)$

<proof>

14 Semantic Trees

abbreviation *closed-branch* :: *partial-pred-denot* \Rightarrow *tree* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

$\text{closed-branch } G T Cs \equiv \text{branch } G T \wedge \text{falsifies}_{c_s} G Cs$

abbreviation(*input*) *open-branch* :: *partial-pred-denot* \Rightarrow *tree* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

$\text{open-branch } G T Cs \equiv \text{branch } G T \wedge \neg \text{falsifies}_{c_s} G Cs$

definition *closed-tree* :: *tree* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**

$\text{closed-tree } T Cs \longleftrightarrow \text{anybranch } T (\lambda b. \text{closed-branch } b T Cs)$
 $\wedge \text{anyinternal } T (\lambda p. \neg \text{falsifies}_{c_s} p Cs)$

15 Herbrand's Theorem

lemma *maximum*:

assumes *asm*: *finite* C

shows $\exists n :: \text{nat}. \forall l \in C. f l \leq n$

<proof>

lemma *extend-preserves-model*:

assumes *f-infnpath*: *wf-infnpath* ($f :: \text{nat} \Rightarrow \text{partial-pred-denot}$)

assumes *C-ground*: *ground* _{l_s} C

assumes *C-sat*: $\neg \text{falsifies}_c (f (\text{Suc } n)) C$

assumes *n-max*: $\forall l \in C. \text{nat-of-fatom } (\text{get-atom } l) \leq n$

shows $eval_c \text{ HFun } (\text{extend } f) \ C$
 $\langle \text{proof} \rangle$

lemma *extend-preserves-model2*:
assumes $f\text{-infnpath}: wf\text{-infnpath } (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes $C\text{-ground}: ground_{1s} \ C$
assumes $fin\text{-}c: finite \ C$
assumes $model\text{-}C: \forall n. \neg falsifies_c (f \ n) \ C$
shows $C\text{-false}: eval_c \ \text{HFun } (\text{extend } f) \ C$
 $\langle \text{proof} \rangle$

lemma *extend-infnpath*:
assumes $f\text{-infnpath}: wf\text{-infnpath } (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes $model\text{-}c: \forall n. \neg falsifies_c (f \ n) \ C$
assumes $fin\text{-}c: finite \ C$
shows $eval_c \ \text{HFun } (\text{extend } f) \ C$
 $\langle \text{proof} \rangle$

If we have a infnpath of partial models, then we have a model.

lemma *infnpath-model*:
assumes $f\text{-infnpath}: wf\text{-infnpath } (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes $model\text{-}cs: \forall n. \neg falsifies_{cs} (f \ n) \ Cs$
assumes $fin\text{-}cs: finite \ Cs$
assumes $fin\text{-}c: \forall C \in Cs. finite \ C$
shows $eval_{cs} \ \text{HFun } (\text{extend } f) \ Cs$
 $\langle \text{proof} \rangle$

fun *deeptree* $:: nat \Rightarrow tree$ **where**
 $deeptree \ 0 = Leaf$
 $| \ \text{deeptree } (Suc \ n) = Branching \ (\text{deeptree } \ n) \ (\text{deeptree } \ n)$

lemma *branch-length*:
assumes $branch \ b \ (\text{deeptree } \ n)$
shows $length \ b = n$
 $\langle \text{proof} \rangle$

lemma *infinity*:
assumes $inj: \forall n :: nat. undiagno \ (\text{diago } \ n) = n$
assumes $all\text{-}tree: \forall n :: nat. (\text{diago } \ n) \in tree$
shows $\neg finite \ tree$
 $\langle \text{proof} \rangle$

lemma *longer-falsifies_l*:
assumes $falsifies_l \ ds \ l$
shows $falsifies_l \ (ds@d) \ l$
 $\langle \text{proof} \rangle$

lemma *longer-falsifies_g*:
assumes $falsifies_g \ ds \ C$

shows $\text{falsifies}_g (ds @ d) C$
 $\langle \text{proof} \rangle$

lemma *longer-falsifies_c*:
assumes $\text{falsifies}_c ds C$
shows $\text{falsifies}_c (ds @ d) C$
 $\langle \text{proof} \rangle$

We use this so that we can apply König's lemma.

lemma *longer-falsifies*:
assumes $\text{falsifies}_{cs} ds Cs$
shows $\text{falsifies}_{cs} (ds @ d) Cs$
 $\langle \text{proof} \rangle$

If all finite semantic trees have an open branch, then the set of clauses has a model.

theorem *herbrand'*:
assumes $\text{openb}: \forall T. \exists G. \text{open-branch } G T Cs$
assumes $\text{finite-cs}: \text{finite } Cs \forall C \in Cs. \text{finite } C$
shows $\exists G. \text{eval}_{cs} \text{HFun } G Cs$
 $\langle \text{proof} \rangle$

lemma *shorter-falsifies_l*:
assumes $\text{falsifies}_l (ds@d) l$
assumes $\text{nat-of-fatom } (\text{get-atom } l) < \text{length } ds$
shows $\text{falsifies}_l ds l$
 $\langle \text{proof} \rangle$

theorem *herbrand'-contra*:
assumes $\text{finite-cs}: \text{finite } Cs \forall C \in Cs. \text{finite } C$
assumes $\text{unsat}: \forall G. \neg \text{eval}_{cs} \text{HFun } G Cs$
shows $\exists T. \forall G. \text{branch } G T \longrightarrow \text{closed-branch } G T Cs$
 $\langle \text{proof} \rangle$

theorem *herbrand*:
assumes $\text{unsat}: \forall G. \neg \text{eval}_{cs} \text{HFun } G Cs$
assumes $\text{finite-cs}: \text{finite } Cs \forall C \in Cs. \text{finite } C$
shows $\exists T. \text{closed-tree } T Cs$
 $\langle \text{proof} \rangle$

end

16 Lifting Lemma

theory *Completeness* **imports** *Resolution* **begin**

locale *unification* =
assumes $\text{unification}: \bigwedge \sigma L. \text{finite } L \Longrightarrow \text{unifier}_{ls} \sigma L \Longrightarrow \exists \vartheta. \text{mgu}_{ls} \vartheta L$

begin

A proof of this assumption is available in `Unification_Theorem.thy` and used in `Completeness_Instance.thy`.

lemma *lifting*:

assumes *fin*: $finite\ C_1 \wedge finite\ C_2$

assumes *apart*: $vars_{l_s}\ C_1 \cap vars_{l_s}\ C_2 = \{\}$

assumes *inst*: $instance-of_{l_s}\ C_1'\ C_1 \wedge instance-of_{l_s}\ C_2'\ C_2$

assumes *appl*: $applicable\ C_1'\ C_2'\ L_1'\ L_2'\ \sigma$

shows $\exists L_1\ L_2\ \tau. applicable\ C_1\ C_2\ L_1\ L_2\ \tau \wedge$
 $instance-of_{l_s}\ (resolution\ C_1'\ C_2'\ L_1'\ L_2'\ \sigma)\ (resolution\ C_1\ C_2\ L_1\ L_2$

$\tau)$

<proof>

17 Completeness

lemma *falsifies_g-empty*:

assumes *falsifies_g* $\square\ C$

shows $C = \{\}$

<proof>

lemma *falsifies_{cs}-empty*:

assumes *falsifies_c* $\square\ C$

shows $C = \{\}$

<proof>

lemma *complements-do-not-falsify'*:

assumes *l1C1'*: $l_1 \in C_1'$

assumes *l2C1'*: $l_2 \in C_1'$

assumes *comp*: $l_1 = l_2^c$

assumes *falsif*: $falsifies_g\ G\ C_1'$

shows *False*

<proof>

lemma *complements-do-not-falsify*:

assumes *l1C1'*: $l_1 \in C_1'$

assumes *l2C1'*: $l_2 \in C_1'$

assumes *fals*: $falsifies_g\ G\ C_1'$

shows $l_1 \neq l_2^c$

<proof>

lemma *other-falsified*:

assumes *C1'-p*: $ground_{l_s}\ C_1' \wedge falsifies_g\ (B@[d])\ C_1'$

assumes *l-p*: $l \in C_1'\ nat-of-fatom\ (get-atom\ l) = length\ B$

assumes *other*: $lo \in C_1'\ lo \neq l$

shows $falsifies_l\ B\ lo$

<proof>

theorem *completeness'*:

assumes *closed-tree* T Cs
assumes $\forall C \in Cs.$ *finite* C
shows $\exists Cs'.$ *resolution-deriv* Cs $Cs' \wedge \{\}$ $\in Cs'$
 \langle *proof* \rangle

theorem *completeness*:

assumes *finite-cs*: *finite* Cs $\forall C \in Cs.$ *finite* C
assumes *unsat*: $\forall (F::\text{hterm fun-denot}) (G::\text{hterm pred-denot}). \neg \text{eval}_{Cs} F G Cs$
shows $\exists Cs'.$ *resolution-deriv* Cs $Cs' \wedge \{\}$ $\in Cs'$
 \langle *proof* \rangle

definition *E-conv* $:: ('a \Rightarrow 'b) \Rightarrow 'a$ *var-denot* $\Rightarrow 'b$ *var-denot* **where**
E-conv *b-of-a* $E \equiv \lambda x. (b\text{-of-}a (E x))$

definition *F-conv* $:: ('a \Rightarrow 'b) \Rightarrow 'a$ *fun-denot* $\Rightarrow 'b$ *fun-denot* **where**
F-conv *b-of-a* $F \equiv \lambda f bs. b\text{-of-}a (F f (\text{map } (\text{inv } b\text{-of-}a) bs))$

definition *G-conv* $:: ('a \Rightarrow 'b) \Rightarrow 'a$ *pred-denot* $\Rightarrow 'b$ *pred-denot* **where**
G-conv *b-of-a* $G \equiv \lambda p bs. (G p (\text{map } (\text{inv } b\text{-of-}a) bs))$

lemma *eval_t-bij*:

assumes *bij* (*b-of-a*: $'a \Rightarrow 'b$)
shows $\text{eval}_t (E\text{-conv } b\text{-of-}a E) (F\text{-conv } b\text{-of-}a F) t = b\text{-of-}a (\text{eval}_t E F t)$
 \langle *proof* \rangle

lemma *eval_{ts}-bij*:

assumes *bij* (*b-of-a*: $'a \Rightarrow 'b$)
shows $G\text{-conv } b\text{-of-}a G p (\text{eval}_{ts} (E\text{-conv } b\text{-of-}a E) (F\text{-conv } b\text{-of-}a F) ts) = G p$
 $(\text{eval}_{ts} E F ts)$
 \langle *proof* \rangle

lemma *eval_l-bij*:

assumes *bij* (*b-of-a*: $'a \Rightarrow 'b$)
shows $\text{eval}_l (E\text{-conv } b\text{-of-}a E) (F\text{-conv } b\text{-of-}a F) (G\text{-conv } b\text{-of-}a G) l = \text{eval}_l E$
 $F G l$
 \langle *proof* \rangle

lemma *eval_c-bij*:

assumes *bij* (*b-of-a*: $'a \Rightarrow 'b$)
shows $\text{eval}_c (F\text{-conv } b\text{-of-}a F) (G\text{-conv } b\text{-of-}a G) C = \text{eval}_c F G C$
 \langle *proof* \rangle

lemma *eval_{cs}-bij*:

assumes *bij* (*b-of-a*: $'a \Rightarrow 'b$)
shows $\text{eval}_{Cs} (F\text{-conv } b\text{-of-}a F) (G\text{-conv } b\text{-of-}a G) Cs \longleftrightarrow \text{eval}_{Cs} F G Cs$
 \langle *proof* \rangle

lemma *countably-inf-bij*:

assumes *inf-a-uni*: *infinite* (*UNIV* :: ('a :: countable) set)
assumes *inf-b-uni*: *infinite* (*UNIV* :: ('b :: countable) set)
shows $\exists b\text{-of-}a :: 'a \Rightarrow 'b$. *bij b-of-a*
 <proof>

lemma *infinite-hterms*: *infinite* (*UNIV* :: *hterm* set)
 <proof>

theorem *completeness-countable*:
assumes *inf-uni*: *infinite* (*UNIV* :: ('u :: countable) set)
assumes *finite-cs*: *finite* *Cs* $\forall C \in Cs$. *finite* *C*
assumes *unsat*: $\forall (F :: 'u \text{ fun-denot}) (G :: 'u \text{ pred-denot})$. $\neg \text{eval}_{cs} F G Cs$
shows $\exists Cs'$. *resolution-deriv* *Cs* *Cs'* $\wedge \{\} \in Cs'$
 <proof>

theorem *completeness-nat*:
assumes *finite-cs*: *finite* *Cs* $\forall C \in Cs$. *finite* *C*
assumes *unsat*: $\forall (F :: \text{nat fun-denot}) (G :: \text{nat pred-denot})$. $\neg \text{eval}_{cs} F G Cs$
shows $\exists Cs'$. *resolution-deriv* *Cs* *Cs'* $\wedge \{\} \in Cs'$
 <proof>

end — unification locale

end

18 Examples

theory *Examples* **imports** *Resolution* **begin**

value *Var* "x"
value *Fun* "one" []
value *Fun* "mul" [Var "y", Var "y"]
value *Fun* "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []]

value *Pos* "greater" [Var "x", Var "y"]
value *Neg* "less" [Var "x", Var "y"]
value *Pos* "less" [Var "x", Var "y"]
value *Pos* "equals"
 [Fun "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []], Var "x"]

fun *F_{nat}* :: *nat fun-denot* **where**
 F_{nat} *f* [*n*, *m*] =
 (if *f* = "add" then *n* + *m* else
 if *f* = "mul" then *n* * *m* else 0)
 | *F_{nat}* *f* [] =
 (if *f* = "one" then 1 else
 if *f* = "zero" then 0 else 0)
 | *F_{nat}* *f* *us* = 0

fun $G_{nat} :: nat\ pred\ denot$ **where**

$G_{nat}\ p\ [x,y] =$
 (if $p = "less" \wedge x < y$ *then* $True$ *else*
 if $p = "greater" \wedge x > y$ *then* $True$ *else*
 if $p = "equals" \wedge x = y$ *then* $True$ *else* $False$)
| $G_{nat}\ p\ us = False$

fun $E_{nat} :: nat\ var\ denot$ **where**

$E_{nat}\ x =$
 (if $x = "x"$ *then* 26 *else*
 if $x = "y"$ *then* 5 *else* 0)

lemma $eval_t\ E_{nat}\ F_{nat}\ (Var\ "x") = 26$

<proof>

lemma $eval_t\ E_{nat}\ F_{nat}\ (Fun\ "one" []) = 1$

<proof>

lemma $eval_t\ E_{nat}\ F_{nat}\ (Fun\ "mul" [Var\ "y", Var\ "y']) = 25$

<proof>

lemma

$eval_t\ E_{nat}\ F_{nat}\ (Fun\ "add" [Fun\ "mul" [Var\ "y", Var\ "y'], Fun\ "one" []]) =$
 26

<proof>

lemma $eval_l\ E_{nat}\ F_{nat}\ G_{nat}\ (Pos\ "greater" [Var\ "x", Var\ "y']) = True$

<proof>

lemma $eval_l\ E_{nat}\ F_{nat}\ G_{nat}\ (Neg\ "less" [Var\ "x", Var\ "y']) = True$

<proof>

lemma $eval_l\ E_{nat}\ F_{nat}\ G_{nat}\ (Pos\ "less" [Var\ "x", Var\ "y']) = False$

<proof>

lemma $eval_l\ E_{nat}\ F_{nat}\ G_{nat}$

$(Pos\ "equals"$
 $[Fun\ "add" [Fun\ "mul" [Var\ "y", Var\ "y'], Fun\ "one" []]$
 $, Var\ "x"]$
 $) = True$

<proof>

definition $PP :: fterm\ literal$ **where**

$PP = Pos\ "P" [Fun\ "c" []]$

definition $PQ :: fterm\ literal$ **where**

$PQ = Pos\ "Q" [Fun\ "d" []]$

definition $NP :: fterm\ literal$ **where**

$NP = Neg\ "P" [Fun\ "c" []]$

definition $NQ :: fterm\ literal$ **where**

$NQ = Neg\ "Q" [Fun\ "d" []]$

theorem *empty-mgu*:
 assumes $unifier_{l_s} \in L$
 shows $mgu_{l_s} \in L$
 <proof>

theorem *unifier-single*: $unifier_{l_s} \sigma \{l\}$
 <proof>

theorem *resolution-rule'*:
 assumes $C_1 \in Cs$
 assumes $C_2 \in Cs$
 assumes *applicable* $C_1 C_2 L_1 L_2 \sigma$
 assumes $C = \{resolution\ C_1\ C_2\ L_1\ L_2\ \sigma\}$
 shows *resolution-step* $Cs\ (Cs \cup C)$
 <proof>

lemma *resolution-example1*:
 $resolution-deriv\ \{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}$
 $\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\},\{PP\},\{\}\}$
 <proof>

definition $Pa :: fterm\ literal\ \mathbf{where}$
 $Pa = Pos\ 'a''\ []$

definition $Na :: fterm\ literal\ \mathbf{where}$
 $Na = Neg\ 'a''\ []$

definition $Pb :: fterm\ literal\ \mathbf{where}$
 $Pb = Pos\ 'b''\ []$

definition $Nb :: fterm\ literal\ \mathbf{where}$
 $Nb = Neg\ 'b''\ []$

definition $Paa :: fterm\ literal\ \mathbf{where}$
 $Paa = Pos\ 'a''\ [Fun\ 'a''\ []]$

definition $Naa :: fterm\ literal\ \mathbf{where}$
 $Naa = Neg\ 'a''\ [Fun\ 'a''\ []]$

definition $Pax :: fterm\ literal\ \mathbf{where}$
 $Pax = Pos\ 'a''\ [Var\ 'x'']$

definition $Nax :: fterm\ literal\ \mathbf{where}$
 $Nax = Neg\ 'a''\ [Var\ 'x'']$

definition $mguPaaPax :: substitution\ \mathbf{where}$
 $mguPaaPax = (\lambda x.\ if\ x = 'x''\ then\ Fun\ 'a''\ []\ else\ Var\ x)$

lemma *mguPaaPax-mgu*: $mgu_{l_s}\ mguPaaPax\ \{Paa,Pax\}$

<proof>

lemma *resolution-example2:*

resolution-deriv $\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}\}$
 $\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}, \{Na, Pb\}, \{Na\}, \{\}\}$

<proof>

lemma *resolution-example1-sem:* $\neg eval_{cs} F G \{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}$

<proof>

lemma *resolution-example2-sem:* $\neg eval_{cs} F G \{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}\}$

<proof>

end

19 The Unification Theorem

theory *Unification-Theorem imports*

First-Order-Terms.Unification Resolution

begin

definition *set-to-list* :: 'a set \Rightarrow 'a list **where**

set-to-list $\equiv inv\ set$

lemma *set-set-to-list:* *finite* *xs* $\Longrightarrow set (set-to-list\ xs) = xs$

<proof>

fun *iterm-to-fterm* :: (fun-sym, var-sym) term \Rightarrow fterm **where**

iterm-to-fterm (Term.Var *x*) = Var *x*

| *iterm-to-fterm* (Term.Fun *f ts*) = Fun *f* (map *iterm-to-fterm ts*)

fun *fterm-to-iterm* :: fterm \Rightarrow (fun-sym, var-sym) term **where**

fterm-to-iterm (Var *x*) = Term.Var *x*

| *fterm-to-iterm* (Fun *f ts*) = Term.Fun *f* (map *fterm-to-iterm ts*)

lemma *iterm-to-fterm-cancel[simp]:* *iterm-to-fterm* (*fterm-to-iterm t*) = *t*

<proof>

lemma *fterm-to-iterm-cancel[simp]:* *fterm-to-iterm* (*iterm-to-fterm t*) = *t*

<proof>

abbreviation(*input*) *fsub-to-isub* :: substitution \Rightarrow (fun-sym, var-sym) subst **where**

fsub-to-isub $\sigma \equiv \lambda x. fterm-to-iterm (\sigma\ x)$

abbreviation(*input*) *isub-to-fsub* :: (fun-sym, var-sym) subst \Rightarrow substitution **where**

isub-to-fsub $\sigma \equiv \lambda x. iterm-to-fterm (\sigma\ x)$

lemma *iterm-to-fterm-subst:* (*iterm-to-fterm t1*) $\cdot_t \sigma = iterm-to-fterm (t1 \cdot (\lambda x. fterm-to-iterm (\sigma\ x)))$

<proof>

lemma *unifiert-unifiers*:

assumes *unifier*_{ts} σ *ts*

shows *fsub-to-isub* $\sigma \in \text{unifiers } (\text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts)$

<proof>

abbreviation(*input*) *get-mgut* :: *fterm list* \Rightarrow *substitution option* **where**

get-mgut *ts* $\equiv \text{map-option } (\text{isub-to-fsub} \circ \text{subst-of}) (\text{unify } (\text{List.product } (\text{map } \text{fterm-to-iterm } ts) (\text{map } \text{fterm-to-iterm } ts)))$ []

lemma *unify-unification*:

assumes $\sigma \in \text{unifiers } (\text{set } E)$

shows $\exists \vartheta. \text{is-imgu } \vartheta (\text{set } E)$

<proof>

lemma *fterm-to-iterm-subst*: $(\text{fterm-to-iterm } t1) \cdot \sigma = \text{fterm-to-iterm } (t1 \cdot_t \text{isub-to-fsub } \sigma)$

<proof>

lemma *unifiers-unifiert*:

assumes $\sigma \in \text{unifiers } (\text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts)$

shows *unifier*_{ts} $(\text{isub-to-fsub } \sigma)$ *ts*

<proof>

lemma *icomf-fcomp*: $\vartheta \circ_s i = \text{fsub-to-isub } (\text{isub-to-fsub } \vartheta \cdot \text{isub-to-fsub } i)$

<proof>

lemma *is-mgu-mgu*_{ts}:

assumes *finite* *ts*

assumes *is-imgu* $\vartheta (\text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts)$

shows *mgu*_{ts} $(\text{isub-to-fsub } \vartheta)$ *ts*

<proof>

lemma *unification'*:

assumes *finite* *ts*

assumes *unifier*_{ts} σ *ts*

shows $\exists \vartheta. \text{mgu}_{ts} \vartheta$ *ts*

<proof>

fun *literal-to-term* :: *fterm literal* \Rightarrow *fterm* **where**

literal-to-term $(\text{Pos } p \text{ } ts) = \text{Fun } ''\text{Pos}'' [\text{Fun } p \text{ } ts]$

| *literal-to-term* $(\text{Neg } p \text{ } ts) = \text{Fun } ''\text{Neg}'' [\text{Fun } p \text{ } ts]$

fun *term-to-literal* :: *fterm* \Rightarrow *fterm literal* **where**

term-to-literal $(\text{Fun } s [\text{Fun } p \text{ } ts]) = (\text{if } s = ''\text{Pos}'' \text{ then } \text{Pos} \text{ else } \text{Neg}) p \text{ } ts$

lemma *term-to-literal-cancel*[*simp*]: *term-to-literal* $(\text{literal-to-term } l) = l$

<proof>

lemma *literal-to-term-sub*: *literal-to-term* ($l \cdot_l \sigma$) = (*literal-to-term* l) $\cdot_t \sigma$
<proof>

lemma *unifier_{l_s}-unifier_{t_s}*:
assumes *unifier_{l_s}* σ L
shows *unifier_{t_s}* σ (*literal-to-term* ‘ L)
<proof>

lemma *unifiert-unifier_{l_s}*:
assumes *unifier_{t_s}* σ (*literal-to-term* ‘ L)
shows *unifier_{l_s}* σ L
<proof>

lemma *mgu_{t_s}-mgu_{l_s}*:
assumes *mgu_{t_s}* ϑ (*literal-to-term* ‘ L)
shows *mgu_{l_s}* ϑ L
<proof>

theorem *unification*:
assumes *fin*: *finite* L
assumes *uni*: *unifier_{l_s}* σ L
shows $\exists \vartheta. \text{mgu}_{l_s} \vartheta$ L
<proof>

end

20 Instance of completeness theorem

theory *Completeness-Instance* imports *Unification-Theorem* *Completeness* begin

interpretation *unification* *<proof>*

thm *lifting*

lemma *lift*:
assumes *fin*: *finite* $C \wedge$ *finite* D
assumes *apart*: *vars_{l_s}* $C \cap$ *vars_{l_s}* $D = \{\}$
assumes *inst₁*: *instance-of_{l_s}* $C' C$
assumes *inst₂*: *instance-of_{l_s}* $D' D$
assumes *appl*: *applicable* $C' D' L' M' \sigma$
shows $\exists L M \tau. \text{applicable } C D L M \tau \wedge$
 $\text{instance-of}_{l_s} (\text{resolution } C' D' L' M' \sigma) (\text{resolution } C D L M \tau)$
<proof>

thm *completeness*

theorem *complete*:

assumes *finite-cs*: *finite Cs* $\forall C \in Cs$. *finite C*

assumes *unsat*: $\forall (F::\text{hterm fun-denot}) (G::\text{hterm pred-denot}) . \neg \text{eval}_{cs} F G Cs$

shows $\exists Cs'$. *resolution-deriv Cs Cs' $\wedge \{\}$ $\in Cs'$*

<proof>

thm *completeness-countable*

theorem *complete-countable*:

assumes *inf-uni*: *infinite (UNIV :: ('u :: countable) set)*

assumes *finite-cs*: *finite Cs* $\forall C \in Cs$. *finite C*

assumes *unsat*: $\forall (F::'u \text{ fun-denot}) (G::'u \text{ pred-denot}) . \neg \text{eval}_{cs} F G Cs$

shows $\exists Cs'$. *resolution-deriv Cs Cs' $\wedge \{\}$ $\in Cs'$*

<proof>

thm *completeness-nat*

theorem *complete-nat*:

assumes *finite-cs*: *finite Cs* $\forall C \in Cs$. *finite C*

assumes *unsat*: $\forall (F::\text{nat fun-denot}) (G::\text{nat pred-denot}) . \neg \text{eval}_{cs} F G Cs$

shows $\exists Cs'$. *resolution-deriv Cs Cs' $\wedge \{\}$ $\in Cs'$*

<proof>

end

References

- [1] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 3rd edition, 2012.
- [2] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1st edition, 1973.
- [3] IsaFoL authors. IsaFoL: Isabelle Formalization of Logic. <https://bitbucket.org/isafol/isafol>.
- [4] A. Leitsch. *The Resolution Calculus*. Texts in theoretical computer science. Springer, 1997.
- [5] A. Schlichtkrull. Formalization of resolution calculus in Isabelle. Msc thesis, Technical University of Denmark, 2015. <https://people.compute.dtu.dk/andschl/Thesis.pdf>.
- [6] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In *ITP 2016*, volume 9807 of *LNCS*. Springer, 2016.
- [7] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 2018.