The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

October 13, 2025

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

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* ⇒ *'t atom* **where**
  get-atom (*Pos p ts*) = (*p*, *ts*)
| *get-atom* (*Neg p ts*) = (*p*, *ts*)

1.1 Ground

fun $ground_t$ :: *fterm* ⇒ *bool* **where**
  $ground_t$ (*Var x*) ⟷ *False*
| $ground_t$ (*Fun f ts*) ⟷ (∀ *t* ∈ *set ts*. $ground_t$ *t*)

abbreviation $ground_{ts}$ :: *fterm list* ⇒ *bool* **where**
  $ground_{ts}$ *ts* ≡ (∀ *t* ∈ *set ts*. $ground_t$ *t*)

abbreviation $ground_l$ :: *fterm literal* ⇒ *bool* **where**
  $ground_l$ *l* ≡ $ground_{ts}$ (*get-terms l*)

abbreviation $ground_{ls}$ :: *fterm literal set* ⇒ *bool* **where**
  $ground_{ls}$ *C* ≡ (∀ *l* ∈ *C*. $ground_l$ *l*)

definition *ground-fatoms* :: *fterm atom set* **where**
  ground-fatoms ≡ {*a*. $ground_{ts}$ (*snd a*)}

lemma $ground_l$*-ground-fatom*:
  assumes $ground_l$ *l*
  shows *get-atom l* ∈ *ground-fatoms*
  ⟨*proof*⟩

1.2 Auxiliary

lemma *infinity*:
  assumes *inj*: $\forall\, n :: nat.\ undiago\ (diago\ n) = n$
  assumes *all-tree*: $\forall\, n :: nat.\ (diago\ n) \in S$
  shows $\neg finite\ S$
⟨*proof*⟩

lemma *inv-into-f-f*:
  assumes *bij-betw f A B*
  assumes $a \in A$
  shows $(inv\text{-}into\ A\ f)\ (f\ a) = a$
⟨*proof*⟩

lemma *f-inv-into-f*:
  assumes *bij-betw f A B*
  assumes $b \in B$
  shows $f\ ((inv\text{-}into\ A\ f)\ b) = b$
⟨*proof*⟩

1.3 Conversions

1.3.1 Conversions - Terms and Herbrand Terms

fun *fterm-of-hterm* :: *hterm* ⇒ *fterm* **where**
  fterm-of-hterm (*HFun p ts*) = *Fun p* (*map fterm-of-hterm ts*)

definition *fterms-of-hterms* :: *hterm list* ⇒ *fterm list* **where**
  fterms-of-hterms ts ≡ *map fterm-of-hterm ts*

fun *hterm-of-fterm* :: *fterm* ⇒ *hterm* **where**
  hterm-of-fterm (*Fun p ts*) = *HFun p* (*map hterm-of-fterm ts*)

definition *hterms-of-fterms* :: *fterm list* ⇒ *hterm list* **where**
hterms-of-fterms ts ≡ *map hterm-of-fterm ts*

lemma *hterm-of-fterm-fterm-of-hterm*[*simp*]: *hterm-of-fterm* (*fterm-of-hterm t*) = *t*
  ⟨*proof*⟩

lemma *hterms-of-fterms-fterms-of-hterms*[*simp*]: *hterms-of-fterms* (*fterms-of-hterms ts*) = *ts*
  ⟨*proof*⟩

lemma *fterm-of-hterm-hterm-of-fterm*[*simp*]:
  assumes $ground_t\ t$
  shows *fterm-of-hterm* (*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*
⟨*proof*⟩

lemma *ground-fterm-of-hterm*: $ground_t$ (*fterm-of-hterm t*)
⟨*proof*⟩

lemma *ground-fterms-of-hterms*: $ground_{ts}$ (*fterms-of-hterms ts*)
⟨*proof*⟩

1.3.2 Conversions - Literals and Herbrand Literals

fun *flit-of-hlit* :: *hterm literal* ⇒ *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* ⇒ *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*)
⟨*proof*⟩

theorem *hlit-of-flit-flit-of-hlit* [*simp*]: *hlit-of-flit* (*flit-of-hlit l*) = *l* ⟨*proof*⟩

theorem *flit-of-hlit-hlit-of-flit* [*simp*]:
assumes $ground_l\ l$
shows *flit-of-hlit* (*hlit-of-flit l*) = *l*
⟨*proof*⟩

lemma *sign-flit-of-hlit*: *sign* (*flit-of-hlit l*) = *sign l* ⟨*proof*⟩

lemma *hlit-of-flit-bij*: *bij-betw hlit-of-flit* {*l*. $ground_l\ l$} *UNIV*
⟨*proof*⟩

lemma *flit-of-hlit-bij*: *bij-betw flit-of-hlit UNIV* {*l*. $ground_l\ l$}
⟨*proof*⟩

1.3.3 Conversions - Atoms and Herbrand Atoms

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

fun *hatom-of-fatom* :: *fterm atom* ⇒ *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*))
⟨*proof*⟩

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

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

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

lemma *fatom-of-hatom-bij*: *bij-betw fatom-of-hatom 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-hatom*:: *hterm atom* ⇒ *nat* **where**
nat-of-hatom ≡ (*SOME f*. *bij f*)

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

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* ⇒ *fterm atom* **where**
fatom-of-nat = (λ*n*. *fatom-of-hatom* (*hatom-of-nat n*))

definition *nat-of-fatom* :: *fterm atom* ⇒ *nat* **where**
nat-of-fatom = (λ*t*. *nat-of-hatom* (*hatom-of-fatom t*))

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

theorem *undiag-diag-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 *bool*s 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* ⇒ *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 ∃ *l r*. *T* = *Branching l r*
  ⟨*proof*⟩

2.2 Paths

fun *path* :: *dir list* ⇒ *tree* ⇒ *bool* **where**
  path [] *T* ⟷ *True*
| *path* (*d*#*ds*) (*Branching T1 T2*) ⟷ (*if d then path ds T1 else path ds T2*)
| *path* - - ⟷ *False*

lemma *path-inv-Leaf*: *path p Leaf* ⟷ *p* = []
  ⟨*proof*⟩

lemma *path-inv-Cons*: *path* (*a*#*ds*) *T* ⟶ (∃ *l r*. *T*=*Branching l r*)
  ⟨*proof*⟩

lemma *path-inv-Branching-Left*: *path* (*Left*#*p*) (*Branching l r*) ⟷ *path p l*
  ⟨*proof*⟩

lemma *path-inv-Branching-Right*: *path* (*Right*#*p*) (*Branching l r*) ⟷ *path p r*
⟨*proof*⟩

lemma *path-inv-Branching*:
  path p (*Branching l r*) ⟷ (*p*=[] ∨ (∃ *a p'*. *p*=*a*#*p'*∧ (*a* ⟶ *path p' l*) ∧ (¬*a* ⟶ *path p' r*))) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *path-prefix*:
  assumes *path* (*ds1*@*ds2*) *T*
  shows *path ds1 T*
⟨*proof*⟩

2.3 Branches

fun *branch* :: *dir list* ⇒ *tree* ⇒ *bool* **where**
  branch [] *Leaf* ⟷ *True*
| *branch* (*d* # *ds*) (*Branching l r*) ⟷ (*if d then branch ds l else branch ds r*)
| *branch* - - ⟷ *False*

lemma *has-branch*: ∃ *b*. *branch b T*
⟨*proof*⟩

lemma *branch-inv-Leaf*: *branch b Leaf* ⟷ *b* = []
⟨*proof*⟩

lemma *branch-inv-Branching-Left*:
  branch (*Left*#*b*) (*Branching l r*) ⟷ *branch b l*
⟨*proof*⟩

lemma *branch-inv-Branching-Right*:
  branch (*Right*#*b*) (*Branching l r*) ⟷ *branch b r*
⟨*proof*⟩

lemma *branch-inv-Branching*:
  branch b (*Branching l r*) ⟷
    (∃ *a b*′. *b*=*a*#*b*′∧ (*a* ⟶ *branch b*′ *l*) ∧ (¬*a* ⟶ *branch b*′ *r*))
⟨*proof*⟩

lemma *branch-inv-Leaf2*:
  T = *Leaf* ⟷ (∀ *b*. *branch b T* ⟶ *b* = [])
⟨*proof*⟩

lemma *branch-is-path*:
  assumes*branch ds T*
  shows *path ds T*
⟨*proof*⟩

lemma *Branching-Leaf-Leaf-Tree*:
  assumes *T* = *Branching T1 T2*
  shows (∃ *B*. *branch* (*B*@[*True*]) *T* ∧ *branch* (*B*@[*False*]) *T*)
⟨*proof*⟩

2.4 Internal Paths

fun *internal* :: *dir list* ⇒ *tree* ⇒ *bool* **where**
  internal [] (*Branching l r*) ⟷ *True*
| *internal* (*d*#*ds*) (*Branching l r*) ⟷ (*if d then internal ds l else internal ds r*)
| *internal* - - ⟷ *False*

lemma *internal-inv-Leaf*: ¬*internal b Leaf* ⟨*proof*⟩

lemma *internal-inv-Branching-Left*:

internal (*Left*#*b*) (*Branching l r*) ⟷ *internal b l* ⟨*proof*⟩

lemma *internal-inv-Branching-Right*:
internal (*Right*#*b*) (*Branching l r*) ⟷ *internal b r*
⟨*proof*⟩

lemma *internal-inv-Branching*:
internal p (*Branching l r*) ⟷ (*p*=[] ∨ (∃ *a p'*. *p*=*a*#*p'*∧ (*a* ⟶ *internal p' l*) ∧
(¬*a* ⟶ *internal p' r*))) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *internal-is-path*:
assumes *internal ds T*
shows *path ds T*
⟨*proof*⟩

lemma *internal-prefix*:
assumes *internal* (*ds1*@*ds2*@[*d*]) *T*
shows *internal ds1 T*
⟨*proof*⟩

lemma *internal-branch*:
assumes *branch* (*ds1*@*ds2*@[*d*]) *T*
shows *internal ds1 T*
⟨*proof*⟩

fun *parent* :: *dir list* ⇒ *dir list* **where**
parent ds = *tl ds*

2.5 Deleting Nodes

fun *delete* :: *dir list* ⇒ *tree* ⇒ *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* ⟨*proof*⟩

lemma *path-delete*:
assumes *path p* (*delete ds T*)
shows *path p T*
⟨*proof*⟩

lemma *branch-delete*:
assumes *branch p* (*delete ds T*)
shows *branch p T* ∨ *p*=*ds*

⟨*proof*⟩

lemma *branch-delete-postfix*:
assumes *path p* (*delete ds T*)
shows ¬(∃ *c cs*. *p* = *ds* @ *c*#*cs*)
⟨*proof*⟩

lemma *treezise-delete*:
assumes *internal p T*
shows *treesize* (*delete p T*) < *treesize T*
⟨*proof*⟩

fun *cutoff* :: (*dir list* ⇒ *bool*) ⇒ *dir list* ⇒ *tree* ⇒ *tree* **where**
cutoff red ds (*Branching* T_1 T_2) =
(*if red ds then Leaf else Branching* (*cutoff red* (*ds*@[*Left*]) T_1) (*cutoff red* (*ds*@[*Right*]) T_2))
| *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 *treesize-cutoff*: *treesize* (*cutoff red ds T*) ≤ *treesize T*
⟨*proof*⟩

abbreviation *anypath* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anypath T P ≡ ∀ *p*. *path p T* ⟶ *P p*

abbreviation *anybranch* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anybranch T P ≡ ∀ *p*. *branch p T* ⟶ *P p*

abbreviation *anyinternal* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anyinternal T P ≡ ∀ *p*. *internal p T* ⟶ *P p*

lemma *cutoff-branch′*:
assumes *anybranch T* (λ*b*. *red*(*ds*@*b*))
shows *anybranch* (*cutoff red ds T*) (λ*b*. *red*(*ds*@*b*))
⟨*proof*⟩

lemma *cutoff-branch*:
assumes *anybranch T* (λ*p*. *red p*)
shows *anybranch* (*cutoff red* [] *T*) (λ*p*. *red p*)
⟨*proof*⟩

lemma *cutoff-internal′*:
assumes *anybranch T* (λ*b*. *red*(*ds*@*b*))
shows *anyinternal* (*cutoff red ds T*) (λ*b*. ¬*red*(*ds*@*b*))
⟨*proof*⟩

lemma *cutoff-internal*:
assumes *anybranch T red*
shows *anyinternal* (*cutoff red* [] *T*) (λ*p*. ¬*red p*)
⟨*proof*⟩

lemma *cutoff-branch-internal'*:
assumes *anybranch T red*
shows *anyinternal* (*cutoff red* [] *T*) (λ*p*. ¬*red p*) ∧ *anybranch* (*cutoff red* [] *T*) (λ*p*. *red p*)
⟨*proof*⟩

lemma *cutoff-branch-internal*:
assumes *anybranch T red*
shows ∃ *T'*. *anyinternal T'* (λ*p*. ¬*red p*) ∧ *anybranch T'* (λ*p*. *red p*)
⟨*proof*⟩

3 Possibly Infinite Trees

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

abbreviation *wf-tree* :: *dir list set* ⇒ *bool* **where**
wf-tree T ≡ (∀ *ds d*. (*ds* @ *d*) ∈ *T* ⟶ *ds* ∈ *T*)

The subtree in with root r

fun *subtree* :: *dir list set* ⇒ *dir list* ⇒ *dir list set* **where**
subtree T r = {*ds* ∈ *T*. ∃ *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 ⊆ *subtree T* (*ds* @ [*Left*]) ∪ *subtree T* (*ds* @ [*Right*]) ∪ {*ds*}
⟨*proof*⟩

3.1 Infinite Paths

abbreviation *wf-infpath* :: (*nat* ⇒ *'a list*) ⇒ *bool* **where**
wf-infpath f ≡ (*f 0* = []) ∧ (∀ *n*. ∃ *a*. *f* (*Suc n*) = (*f n*) @ [*a*])

lemma *infpath-length*:
assumes *wf-infpath f*
shows *length* (*f n*) = *n*
⟨*proof*⟩

lemma *chain-prefix*:
assumes *wf-infpath f*
assumes $n_1 \leq n_2$
shows ∃ *a*. (*f* n_1) @ *a* = (*f* n_2)
⟨*proof*⟩

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-infpath f*
  assumes *smalli*: $i < length\ (f\ n_1)$
  assumes $n_1 n_2$: $n_1 \leq n_2$
  shows $f\ n_1\ !\ i = f\ n_2\ !\ i$
⟨*proof*⟩

4 König's Lemma

lemma *inf-subs*:
  assumes *inf*: ¬*finite*(*subtree T ds*)
  shows ¬*finite*(*subtree T* (*ds* @ [*Left*])) ∨ ¬*finite*(*subtree T* (*ds* @ [*Right*]))
⟨*proof*⟩

fun *buildchain* :: (*dir list* ⇒ *dir list*) ⇒ *nat* ⇒ *dir list* **where**
  buildchain next 0 = []
| *buildchain next* (*Suc n*) = *next* (*buildchain next n*)

lemma *konig*:
  assumes *inf*: ¬*finite T*
  assumes *wellformed*: *wf-tree T*
  shows ∃ *c*. *wf-infpath c* ∧ (∀ *n*. (*c n*) ∈ *T*)
⟨*proof*⟩

end

5 More Terms and Literals

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

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

lemma *cancel-comp1*: $(l^c)^c = l$ ⟨*proof*⟩

lemma *cancel-comp2*:
  assumes *asm*: ${l_1}^c = {l_2}^c$
  shows $l_1 = l_2$
⟨*proof*⟩

lemma *comp-exi1*: ∃ *l'*. $l' = l^c$ ⟨*proof*⟩

lemma *comp-exi2*: ∃ *l*. $l' = l^c$
⟨*proof*⟩

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-pred* $l_1 =$ *get-pred* $l_2 \wedge$ *get-terms* $l_1 =$ *get-terms* $l_2 \longleftrightarrow l_2 = l_1{}^c$
⟨*proof*⟩

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

6 Clauses

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

abbreviation *complementls* :: *'t literal set* ⇒ *'t literal set* (‹-^{C}› [*300*] *300*) **where**

$L^C \equiv$ *complement* ‘ L

lemma *cancel-compls1*: $(L^C)^C = L$
⟨*proof*⟩

lemma *cancel-compls2*:
assumes *asm*: $L_1{}^C = L_2{}^C$
shows $L_1 = L_2$
⟨*proof*⟩

fun $vars_t$ :: *fterm* ⇒ *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* ⇒ *var-sym set* **where**
$vars_{ts}$ *ts* ≡ ($\bigcup t \in$ *set ts*. $vars_t$ t)

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

definition $vars_{ls}$ :: *fterm literal set* ⇒ *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* = {}
⟨*proof*⟩

lemma *ground$_{ls}$-vars$_{ls}$*:
assumes *ground$_{ls}$ L*
shows *vars$_{ls}$ L* = {} ⟨*proof*⟩

lemma *ground-comp*: *ground$_l$* (l^c) ⟷ *ground$_l$ l* ⟨*proof*⟩

lemma *ground-compls*: *ground$_{ls}$* (L^C) ⟷ *ground$_{ls}$ L* ⟨*proof*⟩

7 Semantics

type-synonym *'u fun-denot* = *fun-sym* ⇒ *'u list* ⇒ *'u*
type-synonym *'u pred-denot* = *pred-sym* ⇒ *'u list* ⇒ *bool*
type-synonym *'u var-denot* = *var-sym* ⇒ *'u*

fun *eval$_t$* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *fterm* ⇒ *'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* ⇒ *'u fun-denot* ⇒ *fterm list* ⇒ *'u list* **where**
eval$_{ts}$ E F ts ≡ *map* (*eval$_t$ E F*) *ts*

fun *eval$_l$* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm literal* ⇒ *bool*
where
eval$_l$ E F G (*Pos p ts*) ⟷ *G p* (*eval$_{ts}$ E F ts*)
| *eval$_l$ E F G* (*Neg p ts*) ⟷ ¬*G p* (*eval$_{ts}$ E F ts*)

definition *eval$_c$* :: *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
eval$_c$ F G C ⟷ (∀ *E*. ∃ *l* ∈ *C*. *eval$_l$ E F G l*)

definition *eval$_{cs}$* :: *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm clause set* ⇒ *bool* **where**
eval$_{cs}$ F G Cs ⟷ (∀ *C* ∈ *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*
⟨*proof*⟩

lemma *ground-var-denotts*:
assumes *ground$_{ts}$ ts*
shows *eval$_{ts}$ E F ts* = *eval$_{ts}$ E′ F ts*
⟨*proof*⟩

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

8 Substitutions

type-synonym *substitution* = *var-sym* ⇒ *fterm*

fun *sub* :: *fterm* ⇒ *substitution* ⇒ *fterm* (**infixl** $\langle\cdot_t\rangle$ 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* ⇒ *substitution* ⇒ *fterm list* (**infixl** $\langle\cdot_{ts}\rangle$ 55) **where**
  $ts\ \cdot_{ts}\ \sigma \equiv (map\ (\lambda t.\ t\ \cdot_t\ \sigma)\ ts)$

fun *subl* :: *fterm literal* ⇒ *substitution* ⇒ *fterm literal* (**infixl** $\langle\cdot_l\rangle$ 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* ⇒ *substitution* ⇒ *fterm literal set* (**infixl** $\langle\cdot_{ls}\rangle$ 55) **where**
  $L\ \cdot_{ls}\ \sigma \equiv (\lambda l.\ l\ \cdot_l\ \sigma)\ `\ L$

lemma *subls-def2*: $L\ \cdot_{ls}\ \sigma = \{l\ \cdot_l\ \sigma | l.\ l \in L\}$ ⟨*proof*⟩

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

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

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

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

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

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

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

definition *var-renaming-of* :: *fterm clause* ⇒ *fterm clause* ⇒ *bool* **where**
var-renaming-of C_1 C_2 ⟷ *instance-of*$_{ls}$ C_1 C_2 ∧ *instance-of*$_{ls}$ C_2 C_1

8.1 The Empty Substitution

abbreviation ε :: *substitution* **where**
$\varepsilon \equiv$ *Var*

lemma *empty-subt*: $(t :: fterm) \cdot_t \varepsilon = t$
⟨*proof*⟩

lemma *empty-subts*: $ts \cdot_{ts} \varepsilon = ts$
⟨*proof*⟩

lemma *empty-subl*: $l \cdot_l \varepsilon = l$
⟨*proof*⟩

lemma *empty-subls*: $L \cdot_{ls} \varepsilon = L$
⟨*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*$_{ls}$*-self*: *instance-of*$_{ls}$ 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*$_{ls}$*-subls*:

assumes *ground*: $ground_{ls}\ L$
shows $L \cdot_{ls} \sigma = L$
⟨*proof*⟩

8.3 Composition

definition *composition* :: *substitution* ⇒ *substitution* ⇒ *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 $instance\text{-}of_t$*-trans* :
assumes t_{12}: $instance\text{-}of_t\ t_1\ t_2$
assumes t_{23}: $instance\text{-}of_t\ t_2\ t_3$
shows $instance\text{-}of_t\ t_1\ t_3$
⟨*proof*⟩

lemma $instance\text{-}of_{ts}$*-trans* :
assumes ts_{12}: $instance\text{-}of_{ts}\ ts_1\ ts_2$
assumes ts_{23}: $instance\text{-}of_{ts}\ ts_2\ ts_3$
shows $instance\text{-}of_{ts}\ ts_1\ ts_3$
⟨*proof*⟩

lemma $instance\text{-}of_l$*-trans* :
assumes l_{12}: $instance\text{-}of_l\ l_1\ l_2$
assumes l_{23}: $instance\text{-}of_l\ l_2\ l_3$
shows $instance\text{-}of_l\ l_1\ l_3$
⟨*proof*⟩

lemma *instance-of$_{ls}$-trans* :
assumes L_{12}: *instance-of$_{ls}$* L_1 L_2
assumes L_{23}: *instance-of$_{ls}$* L_2 L_3
shows *instance-of$_{ls}$* L_1 L_3
⟨*proof*⟩

8.4 Merging substitutions

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

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

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

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

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

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

8.5 Standardizing apart

abbreviation std_1 :: *fterm clause* ⇒ *fterm clause* **where**
$std_1\ C \equiv C \cdot_{ls} (\lambda x.\ Var\ (''1'' @ x))$

abbreviation std_2 :: *fterm clause* ⇒ *fterm clause* **where**
$std_2\ C \equiv C \cdot_{ls} (\lambda x.\ Var\ (''2'' @ x))$

lemma *std-apart-apart''*:
assumes $x \in vars_t\ \ (t \cdot_t (\lambda x{::}char\ list.\ Var\ (y\ @\ x)))$
shows $\exists x'.\ x = y@x'$
⟨*proof*⟩

lemma *std-apart-apart'*:
assumes $x \in vars_l\ (l \cdot_l (\lambda x.\ Var\ \ (y@x)))$
shows $\exists x'.\ x = y@x'$
⟨*proof*⟩

lemma *std-apart-apart*: $vars_{ls}\ (std_1\ C_1) \cap vars_{ls}\ (std_2\ C_2) = \{\}$
⟨*proof*⟩

lemma *std-apart-instance-of$_{ls}$1*: $instance\text{-}of_{ls}\ C_1\ (std_1\ C_1)$
⟨*proof*⟩

lemma *std-apart-instance-of$_{ls}$2*: $instance\text{-}of_{ls}\ C2\ (std_2\ C2)$
⟨*proof*⟩

9 Unifiers

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

definition $unifier_{ls}$ :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
$unifier_{ls}\ \sigma\ L \longleftrightarrow (\exists l'.\ \forall l \in L.\ l \cdot_l \sigma = l')$

lemma *unif-sub*:
assumes *unif*: $unifier_{ls}\ \sigma\ L$
assumes *nonempty*: $L \neq \{\}$
shows $\exists l.\ subls\ L\ \sigma = \{subl\ l\ \sigma\}$
⟨*proof*⟩

lemma *unifiert-def2*:
assumes *L-elem*: $ts \neq \{\}$
shows $unifier_{ts}\ \sigma\ ts \longleftrightarrow (\exists l.\ (\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{l\})$
⟨*proof*⟩

lemma *unifier$_{ls}$-def2*:
assumes *L-elem*: $L \neq \{\}$
shows $unifier_{ls}\ \sigma\ L \longleftrightarrow (\exists l.\ L \cdot_{ls} \sigma = \{l\})$
⟨*proof*⟩

lemma *ground$_{ls}$-unif-singleton*:
assumes *ground$_{ls}$*: $ground_{ls}\ L$
assumes *unif*: $unifier_{ls}\ \sigma'\ L$
assumes *empt*: $L \neq \{\}$
shows $\exists l.\ L = \{l\}$
⟨*proof*⟩

definition *unifiablets* :: *fterm set* ⇒ *bool* **where**
unifiablets fs ⟷ (∃ σ. *unifier*$_{ts}$ σ *fs*)

definition *unifiablels* :: *fterm literal set* ⇒ *bool* **where**
unifiablels L ⟷ (∃ σ. *unifier*$_{ls}$ σ *L*)

lemma *unifier-comp*[*simp*]: *unifier*$_{ls}$ σ (L^C) ⟷ *unifier*$_{ls}$ σ *L*
⟨*proof*⟩

lemma *unifier-sub1*:
assumes *unifier*$_{ls}$ σ *L*
assumes $L' \subseteq L$
shows *unifier*$_{ls}$ σ L'
⟨*proof*⟩

lemma *unifier-sub2*:
assumes *asm*: *unifier*$_{ls}$ σ ($L_1 \cup L_2$)
shows *unifier*$_{ls}$ σ L_1 ∧ *unifier*$_{ls}$ σ L_2
⟨*proof*⟩

9.1 Most General Unifiers

definition *mgu*$_{ts}$ :: *substitution* ⇒ *fterm set* ⇒ *bool* **where**
mgu$_{ts}$ σ *ts* ⟷ *unifier*$_{ts}$ σ *ts* ∧ (∀ *u*. *unifier*$_{ts}$ *u ts* ⟶ (∃ *i*. *u* = σ · *i*))

definition *mgu*$_{ls}$ :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
mgu$_{ls}$ σ *L* ⟷ *unifier*$_{ls}$ σ *L* ∧ (∀ *u*. *unifier*$_{ls}$ *u L* ⟶ (∃ *i*. *u* = σ · *i*))

10 Resolution

definition *applicable* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *bool* **where**
applicable C_1 C_2 L_1 L_2 σ ⟷
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
∧ *vars*$_{ls}$ C_1 ∩ *vars*$_{ls}$ C_2 = {}
∧ $L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
∧ *mgu*$_{ls}$ σ ($L_1 \cup {L_2}^C$)

definition *mresolution* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *fterm clause* **where**
mresolution C_1 C_2 L_1 L_2 σ = ((C_1 ·$_{ls}$ σ)− (L_1 ·$_{ls}$ σ)) ∪ ((C_2 ·$_{ls}$ σ) − (L_2 ·$_{ls}$ σ))

definition *resolution* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *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* ⇒ *fterm clause set* ⇒ *bool* **where**
mresolution-rule:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *applicable* $C_1\ C_2\ L_1\ L_2\ \sigma \Longrightarrow$
mresolution-step $Cs\ (Cs \cup \{$*mresolution* $C_1\ C_2\ L_1\ L_2\ \sigma\})$
| *standardize-apart*:
$C \in Cs \Longrightarrow$ *var-renaming-of* $C\ C' \Longrightarrow$ *mresolution-step* $Cs\ (Cs \cup \{C'\})$

inductive *resolution-step* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-rule:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *applicable* $C_1\ C_2\ L_1\ L_2\ \sigma \Longrightarrow$
resolution-step $Cs\ (Cs \cup \{$*resolution* $C_1\ C_2\ L_1\ L_2\ \sigma\})$
| *standardize-apart*:
$C \in Cs \Longrightarrow$ *var-renaming-of* $C\ C' \Longrightarrow$ *resolution-step* $Cs\ (Cs \cup \{C'\})$

definition *mresolution-deriv* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
mresolution-deriv = *rtranclp mresolution-step*

definition *resolution-deriv* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-deriv = *rtranclp resolution-step*

11 Soundness

definition *evalsub* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *substitution* ⇒ *'u var-denot* **where**
evalsub $E\ F\ \sigma = eval_t\ E\ F \circ \sigma$

lemma *substitutiont*: $eval_t\ E\ F\ (t \cdot_t \sigma) = eval_t\ (evalsub\ E\ F\ \sigma)\ F\ t$
⟨*proof*⟩

lemma *substitutionts*: $eval_{ts}\ E\ F\ (ts \cdot_{ts} \sigma) = eval_{ts}\ (evalsub\ E\ F\ \sigma)\ F\ ts$
⟨*proof*⟩

lemma *substitution*: $eval_l\ E\ F\ G\ (l \cdot_l \sigma) \longleftrightarrow eval_l\ (evalsub\ E\ F\ \sigma)\ F\ G\ l$
⟨*proof*⟩

lemma *subst-sound*:
assumes *asm*: $eval_c\ F\ G\ C$
shows $eval_c\ F\ G\ (C \cdot_{ls} \sigma)$
⟨*proof*⟩

lemma *simple-resolution-sound*:
assumes $C_1 sat$: $eval_c\ F\ G\ C_1$
assumes $C_2 sat$: $eval_c\ F\ G\ C_2$
assumes $l_1 inc_1$: $l_1 \in C_1$
assumes $l_2 inc_2$: $l_2 \in C_2$
assumes *comp*: ${l_1}^c = l_2$
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 $\neg eval_{cs}$ *F G Cs*
⟨*proof*⟩

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 (*eval*$_t$ *E HFun t*) = *hterm-of-fterm t*
⟨*proof*⟩

lemma *eval-ground*$_{ts}$:
assumes *ground*$_{ts}$ *ts*
shows (*eval*$_{ts}$ *E HFun ts*) = *hterms-of-fterms ts*
⟨*proof*⟩

lemma *eval*$_l$*-ground*$_{ts}$:
assumes *asm*: *ground*$_{ts}$ *ts*
shows *eval*$_l$ *E HFun G* (*Pos P ts*) $\longleftrightarrow$ *G P* (*hterms-of-fterms ts*)
⟨*proof*⟩

13 Partial Interpretations

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

definition *falsifies*$_l$ :: *partial-pred-denot* $\Rightarrow$ *fterm literal* $\Rightarrow$ *bool* **where**
falsifies$_l$ *G l* $\longleftrightarrow$
ground$_l$ *l*
$\wedge$ (*let i* = *nat-of-fatom* (*get-atom l*) *in*
i < *length G* $\wedge$ *G* ! *i* = (¬*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**
falsifies$_g$ *G C* ≡ *ground*$_{ls}$ *C* $\wedge$ ($\forall$ *l* $\in$ *C*. *falsifies*$_l$ *G l*)

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

abbreviation $falsifies_{cs}$ :: *partial-pred-denot* ⇒ *fterm clause set* ⇒ *bool* **where**
$falsifies_{cs}\ G\ Cs \equiv (\exists\, C \in Cs.\ falsifies_c\ G\ C)$

abbreviation *extend* :: (*nat* ⇒ *partial-pred-denot*) ⇒ *hterm pred-denot* **where**
extend f P ts ≡ (
let n = *nat-of-hatom* (*P*, *ts*) *in*
f (*Suc n*) ! *n*
)

fun *sub-of-denot* :: *hterm var-denot* ⇒ *substitution* **where**
sub-of-denot E = *fterm-of-hterm* ∘ *E*

lemma *ground-sub-of-denott*: $ground_t\ (t \cdot_t (sub\text{-}of\text{-}denot\ E))$
⟨*proof*⟩

lemma *ground-sub-of-denotts*: $ground_{ts}\ (ts \cdot_{ts} sub\text{-}of\text{-}denot\ E)$
⟨*proof*⟩

lemma *ground-sub-of-denotl*: $ground_l\ (l \cdot_l sub\text{-}of\text{-}denot\ E)$
⟨*proof*⟩

lemma *sub-of-denot-equivx*: $eval_t\ E\ HFun\ (sub\text{-}of\text{-}denot\ E\ x) = E\ x$
⟨*proof*⟩

lemma *sub-of-denot-equivt*:
$eval_t\ E\ HFun\ (t \cdot_t (sub\text{-}of\text{-}denot\ E)) = eval_t\ E\ HFun\ t$
⟨*proof*⟩

lemma *sub-of-denot-equivts*: $eval_{ts}\ E\ HFun\ (ts \cdot_{ts} (sub\text{-}of\text{-}denot\ E)) = eval_{ts}\ E\ HFun\ ts$
⟨*proof*⟩

lemma *sub-of-denot-equivl*: $eval_l\ E\ HFun\ G\ (l \cdot_l sub\text{-}of\text{-}denot\ E) \longleftrightarrow eval_l\ E\ HFun\ G\ l$
⟨*proof*⟩

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

lemma *sub-of-denot-equiv-ground′*:
$eval_l\ E\ HFun\ G\ l = eval_l\ E\ HFun\ G\ (l \cdot_l sub\text{-}of\text{-}denot\ E) \wedge ground_l\ (l \cdot_l sub\text{-}of\text{-}denot\ E)$
⟨*proof*⟩

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

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

shows $falsifies_c\ G\ C$
⟨*proof*⟩

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

lemma *sub-of-denot-equiv-ground*:
$((\exists\, l \in C.\ eval_l\ E\ HFun\ G\ l) \longleftrightarrow (\exists\, l \in C \cdot_{ls} sub\text{-}of\text{-}denot\ E.\ eval_l\ E\ HFun\ G\ l))$
$\wedge\ ground_{ls}\ (C \cdot_{ls} sub\text{-}of\text{-}denot\ E)$
⟨*proof*⟩

lemma std_1-*falsifies*: $falsifies_c\ G\ C_1 \longleftrightarrow falsifies_c\ G\ (std_1\ C_1)$
⟨*proof*⟩

lemma std_2-*falsifies*: $falsifies_c\ G\ C_2 \longleftrightarrow falsifies_c\ G\ (std_2\ C_2)$
⟨*proof*⟩

lemma std_1-*renames*: *var-renaming-of* $C_1\ (std_1\ C_1)$
⟨*proof*⟩

lemma std_2-*renames*: *var-renaming-of* $C_2\ (std_2\ C_2)$
⟨*proof*⟩

14 Semantic Trees

abbreviation *closed-branch* :: *partial-pred-denot* ⇒ *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
closed-branch G T Cs ≡ *branch G T* ∧ $falsifies_{cs}\ G\ Cs$

abbreviation(*input*) *open-branch* :: *partial-pred-denot* ⇒ *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
open-branch G T Cs ≡ *branch G T* ∧ ¬$falsifies_{cs}\ G\ Cs$

definition *closed-tree* :: *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
closed-tree T Cs ⟷ *anybranch T* (λ*b*. *closed-branch b T Cs*)
∧ *anyinternal T* (λ*p*. ¬$falsifies_{cs}\ p\ Cs$)

15 Herbrand's Theorem

lemma *maximum*:
assumes *asm*: *finite C*
shows ∃ *n* :: *nat*. ∀ *l* ∈ *C*. *f l* ≤ *n*
⟨*proof*⟩

lemma *extend-preserves-model*:
assumes *f-infpath*: *wf-infpath* (*f* :: *nat* ⇒ *partial-pred-denot*)
assumes *C-ground*: $ground_{ls}\ C$
assumes *C-sat*: ¬$falsifies_c\ (f\ (Suc\ n))\ C$

assumes *n-max*: $\forall\, l{\in}C.\ nat\text{-}of\text{-}fatom\ (get\text{-}atom\ l) \leq n$
shows $eval_c\ HFun\ (extend\ f)\ C$
⟨*proof*⟩

lemma *extend-preserves-model2*:
assumes *f-infpath*: *wf-infpath* $(f :: nat \Rightarrow partial\text{-}pred\text{-}denot)$
assumes *C-ground*: $ground_{ls}\ C$
assumes *fin-c*: *finite C*
assumes *model-C*: $\forall\, n.\ \neg falsifies_c\ (f\ n)\ C$
shows *C-false*: $eval_c\ HFun\ (extend\ f)\ C$
⟨*proof*⟩

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

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

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

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

lemma *branch-length*:
assumes *branch b* (*deeptree n*)
shows *length b* = *n*
⟨*proof*⟩

lemma *infinity*:
assumes *inj*: $\forall\, n :: nat.\ undiago\ (diago\ n) = n$
assumes *all-tree*: $\forall\, n :: nat.\ (diago\ n) \in tree$
shows ¬*finite tree*
⟨*proof*⟩

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

lemma $longer\text{-}falsifies_g$:

assumes $falsifies_g\ ds\ C$
shows $falsifies_g\ (ds\ @\ d)\ C$
⟨*proof*⟩

lemma *longer-falsifies*$_c$:
assumes $falsifies_c\ ds\ C$
shows $falsifies_c\ (ds\ @\ d)\ C$
⟨*proof*⟩

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

lemma *longer-falsifies*:
assumes $falsifies_{cs}\ ds\ Cs$
shows $falsifies_{cs}\ (ds\ @\ d)\ Cs$
⟨*proof*⟩

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

theorem *herbrand′*:
assumes *openb*: $\forall\, T.\ \exists\, G.\ open\text{-}branch\ G\ T\ Cs$
assumes *finite-cs*: $finite\ Cs\ \forall\, C{\in}Cs.\ finite\ C$
shows $\exists\, G.\ eval_{cs}\ HFun\ G\ Cs$
⟨*proof*⟩

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

theorem *herbrand′-contra*:
assumes *finite-cs*: $finite\ Cs\ \forall\, C{\in}Cs.\ finite\ C$
assumes *unsat*: $\forall\, G.\ \neg eval_{cs}\ HFun\ G\ Cs$
shows $\exists\, T.\ \forall\, G.\ branch\ G\ T \longrightarrow closed\text{-}branch\ G\ T\ Cs$
⟨*proof*⟩

theorem *herbrand*:
assumes *unsat*: $\forall\, G.\ \neg eval_{cs}\ HFun\ G\ Cs$
assumes *finite-cs*: $finite\ Cs\ \forall\, C{\in}Cs.\ finite\ C$
shows $\exists\, T.\ closed\text{-}tree\ T\ Cs$
⟨*proof*⟩

end

16 Lifting Lemma

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

locale *unification* =

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

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

lemma *lifting*:
assumes *fin*: $\mathit{finite}\ C_1 \wedge \mathit{finite}\ C_2$
assumes *apart*: $\mathit{vars}_{ls}\ C_1 \cap \mathit{vars}_{ls}\ C_2 = \{\}$
assumes *inst*: $\mathit{instance\text{-}of}_{ls}\ C_1'\ C_1 \wedge \mathit{instance\text{-}of}_{ls}\ C_2'\ C_2$
assumes *appl*: $\mathit{applicable}\ C_1'\ C_2'\ L_1'\ L_2'\ \sigma$
shows $\exists L_1\ L_2\ \tau.\ \mathit{applicable}\ C_1\ C_2\ L_1\ L_2\ \tau \wedge$
$\mathit{instance\text{-}of}_{ls}\ (\mathit{resolution}\ C_1'\ C_2'\ L_1'\ L_2'\ \sigma)\ (\mathit{resolution}\ C_1\ C_2\ L_1\ L_2\ \tau)$
⟨*proof*⟩

17 Completeness

lemma $\mathit{falsifies}_g$-*empty*:
assumes $\mathit{falsifies}_g\ []\ C$
shows $C = \{\}$
⟨*proof*⟩

lemma $\mathit{falsifies}_{cs}$-*empty*:
assumes $\mathit{falsifies}_c\ []\ C$
shows $C = \{\}$
⟨*proof*⟩

lemma *complements-do-not-falsify'*:
assumes *l1C1'*: $l_1 \in C_1'$
assumes l_2*C1'*: $l_2 \in C_1'$
assumes *comp*: $l_1 = {l_2}^c$
assumes *falsif*: $\mathit{falsifies}_g\ G\ C_1'$
shows *False*
⟨*proof*⟩

lemma *complements-do-not-falsify*:
assumes *l1C1'*: $l_1 \in C_1'$
assumes l_2*C1'*: $l_2 \in C_1'$
assumes *fals*: $\mathit{falsifies}_g\ G\ C_1'$
shows $l_1 \neq {l_2}^c$
⟨*proof*⟩

lemma *other-falsified*:
assumes *C1'-p*: $\mathit{ground}_{ls}\ C_1' \wedge \mathit{falsifies}_g\ (B@[d])\ C_1'$
assumes *l-p*: $l \in C_1'\ \mathit{nat\text{-}of\text{-}fatom}\ (\mathit{get\text{-}atom}\ l) = \mathit{length}\ B$
assumes *other*: $\mathit{lo} \in C_1'\ \mathit{lo} \neq l$
shows $\mathit{falsifies}_l\ B\ \mathit{lo}$
⟨*proof*⟩

theorem *completeness′*:
assumes *closed-tree T Cs*
assumes $\forall\, C{\in}Cs.$ *finite C*
shows $\exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$
⟨*proof*⟩

theorem *completeness*:
assumes *finite-cs*: *finite Cs* $\forall\, C{\in}Cs.$ *finite C*
assumes *unsat*: $\forall$ (*F*::*hterm fun-denot*) (*G*::*hterm pred-denot*) . $\neg$*eval*$_{cs}$ *F G Cs*
shows $\exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$
⟨*proof*⟩

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-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-of-a* (*F f* (*map* (*inv b-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* (*map* (*inv b-of-a*) *bs*))

lemma *eval*$_t$*-bij*:
assumes *bij* (*b-of-a*::*′a* $\Rightarrow$ *′b*)
shows*eval*$_t$ (*E-conv b-of-a E*) (*F-conv b-of-a F*) *t* = *b-of-a* (*eval*$_t$ *E F t*)
⟨*proof*⟩

lemma *eval*$_{ts}$*-bij*:
assumes *bij* (*b-of-a*::*′a* $\Rightarrow$ *′b*)
shows *G-conv b-of-a G p* (*eval*$_{ts}$ (*E-conv b-of-a E*) (*F-conv b-of-a F*) *ts*) = *G p* (*eval*$_{ts}$ *E F ts*)
⟨*proof*⟩

lemma *eval*$_l$*-bij*:
assumes *bij* (*b-of-a*::*′a* $\Rightarrow$ *′b*)
shows *eval*$_l$ (*E-conv b-of-a E*) (*F-conv b-of-a F*) (*G-conv b-of-a G*) *l* = *eval*$_l$ *E F G l*
⟨*proof*⟩

lemma *eval*$_c$*-bij*:
assumes *bij* (*b-of-a*::*′a* $\Rightarrow$ *′b*)
shows *eval*$_c$ (*F-conv b-of-a F*) (*G-conv b-of-a G*) *C* = *eval*$_c$ *F G C*
⟨*proof*⟩

lemma *eval*$_{cs}$*-bij*:
assumes *bij* (*b-of-a*::*′a* $\Rightarrow$ *′b*)
shows *eval*$_{cs}$ (*F-conv b-of-a F*) (*G-conv b-of-a G*) *Cs* $\longleftrightarrow$ *eval*$_{cs}$ *F G Cs*
⟨*proof*⟩

lemma *countably-inf-bij*:
  assumes *inf-a-uni*: *infinite* (*UNIV* :: ('*a* ::*countable*) *set*)
  assumes *inf-b-uni*: *infinite* (*UNIV* :: ('*b* ::*countable*) *set*)
  shows ∃ *b-of-a* :: '*a* ⇒ '*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* ∀ *C*∈*Cs*. *finite C*
  assumes *unsat*: ∀ (*F*::'*u fun-denot*) (*G*::'*u pred-denot*). ¬*eval*$_{cs}$ *F G Cs*
  shows ∃ *Cs'*. *resolution-deriv Cs Cs'* ∧ {} ∈ *Cs'*
⟨*proof*⟩

theorem *completeness-nat*:
  assumes *finite-cs*: *finite Cs* ∀ *C*∈*Cs*. *finite C*
  assumes *unsat*: ∀ (*F*::*nat fun-denot*) (*G*::*nat pred-denot*) . ¬*eval*$_{cs}$ *F G Cs*
  shows ∃ *Cs'*. *resolution-deriv Cs Cs'* ∧ {} ∈ *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*'' ∧ *x* < *y then True else*
     if p = ''*greater*'' ∧ *x* > *y then True else*
     if p = ''*equals*'' ∧ *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 $unifier_{ls}\ \varepsilon\ L$
  shows $mgu_{ls}\ \varepsilon\ L$
⟨*proof*⟩

theorem *unifier-single*: $unifier_{ls}\ \sigma\ \{l\}$
⟨*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* **where**
  Pa = *Pos* ''*a*'' []

definition *Na* :: *fterm literal* **where**
  Na = *Neg* ''*a*'' []

definition *Pb* :: *fterm literal* **where**
  Pb = *Pos* ''*b*'' []

definition *Nb* :: *fterm literal* **where**
  Nb = *Neg* ''*b*'' []

definition *Paa* :: *fterm literal* **where**
  Paa = *Pos* ''*a*'' [*Fun* ''*a*'' []]

definition *Naa* :: *fterm literal* **where**
  Naa = *Neg* ''*a*'' [*Fun* ''*a*'' []]

definition *Pax* :: *fterm literal* **where**
  Pax = *Pos* ''*a*'' [*Var* ''*x*'']

definition *Nax* :: *fterm literal* **where**
  Nax = *Neg* ''*a*'' [*Var* ''*x*'']

definition *mguPaaPax* :: *substitution* **where**
  mguPaaPax = (λ*x*. *if* *x* = ''*x*'' *then* *Fun* ''*a*'' [] *else* *Var* *x*)

lemma *mguPaaPax-mgu*: mgu_{ls} *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*: ¬$eval_{cs}$ *F G* {{*NP*, *PQ*}, {*NQ*}, {*PP*, *PQ*}}
⟨*proof*⟩

lemma *resolution-example2-sem*: ¬$eval_{cs}$ *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* ⇒ *'a list* **where**
set-to-list ≡ *inv set*

lemma *set-set-to-list*: *finite xs* ⟹ *set* (*set-to-list xs*) = *xs*
⟨*proof*⟩

fun *iterm-to-fterm* :: (*fun-sym*, *var-sym*) *term* ⇒ *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* ⇒ (*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* ⇒ (*fun-sym*, *var-sym*) *subst* **where**
fsub-to-isub σ ≡ λ*x*. *fterm-to-iterm* (σ *x*)

abbreviation(*input*) *isub-to-fsub* :: (*fun-sym*, *var-sym*) *subst* ⇒ *substitution* **where**
isub-to-fsub σ ≡ λ*x*. *iterm-to-fterm* (σ *x*)

lemma *iterm-to-fterm-subt*: (*iterm-to-fterm t1*) ·$_t$ σ = *iterm-to-fterm* (*t1* · (λ*x*.

fterm-to-iterm (σ *x*)))
⟨*proof*⟩

lemma *unifiert-unifiers*:
assumes $unifier_{ts}$ σ *ts*
shows *fsub-to-isub* $\sigma \in$ *unifiers* (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
⟨*proof*⟩

abbreviation(*input*) *get-mgut* :: *fterm list* ⇒ *substitution option* **where**
get-mgut ts ≡ *map-option* (*isub-to-fsub* ∘ *subst-of*) (*unify* (*List.product* (*map fterm-to-iterm ts*) (*map fterm-to-iterm ts*)) [])

lemma *unify-unification*:
assumes $\sigma \in$ *unifiers* (*set E*)
shows $\exists \vartheta$. *is-imgu* ϑ (*set E*)
⟨*proof*⟩

lemma *fterm-to-iterm-subst*: (*fterm-to-iterm t1*) · σ =*fterm-to-iterm* (*t1* $\cdot_t$ *isub-to-fsub* σ)
⟨*proof*⟩

lemma *unifiers-unifiert*:
assumes $\sigma \in$ *unifiers* (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
shows $unifier_{ts}$ (*isub-to-fsub* σ) *ts*
⟨*proof*⟩

lemma *icomp-fcomp*: $\vartheta \circ_s i$ = *fsub-to-isub* (*isub-to-fsub* ϑ · *isub-to-fsub i*)
⟨*proof*⟩

lemma *is-mgu-*mgu_{ts}:
assumes *finite ts*
assumes *is-imgu* ϑ (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
shows mgu_{ts} (*isub-to-fsub* ϑ) *ts*
⟨*proof*⟩

lemma *unification′*:
assumes *finite ts*
assumes $unifier_{ts}$ σ *ts*
shows $\exists \vartheta$. mgu_{ts} ϑ *ts*
⟨*proof*⟩

fun *literal-to-term* :: *fterm literal* ⇒ *fterm* **where**
literal-to-term (*Pos p ts*) = *Fun* ″*Pos*″ [*Fun p ts*]
| *literal-to-term* (*Neg p ts*) = *Fun* ″*Neg*″ [*Fun p ts*]

fun *term-to-literal* :: *fterm* ⇒ *fterm literal* **where**
term-to-literal (*Fun s* [*Fun p ts*]) = (*if s*=″*Pos*″ *then Pos else Neg*) *p ts*

lemma *term-to-literal-cancel*[*simp*]: *term-to-literal* (*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 $unifier_{ls}$-$unifier_{ts}$:
assumes $unifier_{ls}\ \sigma\ L$
shows $unifier_{ts}\ \sigma$ (*literal-to-term* ‘ L)
⟨*proof*⟩

lemma *unifiert-unifier*$_{ls}$:
assumes $unifier_{ts}\ \sigma$ (*literal-to-term* ‘ L)
shows $unifier_{ls}\ \sigma\ L$
⟨*proof*⟩

lemma mgu_{ts}-mgu_{ls}:
assumes $mgu_{ts}\ \vartheta$ (*literal-to-term* ‘ L)
shows $mgu_{ls}\ \vartheta\ L$
⟨*proof*⟩

theorem *unification*:
assumes *fin*: *finite* L
assumes *uni*: $unifier_{ls}\ \sigma\ L$
shows $\exists \vartheta.\ mgu_{ls}\ \vartheta\ L$
⟨*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*: $vars_{ls}\ C \cap vars_{ls}\ D = \{\}$
assumes $inst_1$: *instance-of*$_{ls}$ C' C
assumes $inst_2$: *instance-of*$_{ls}$ D' D
assumes *appl*: *applicable* C' D' L' M' σ
shows $\exists L\ M\ \tau.$ *applicable* C D L M $\tau \wedge$
instance-of$_{ls}$ (*resolution* C' D' L' M' σ) (*resolution* C D L M τ)
⟨*proof*⟩

thm *completeness*

theorem *complete*:
assumes *finite-cs*: *finite Cs* $\forall$ *C*$\in$*Cs*. *finite C*
assumes *unsat*: $\forall$ (*F*::*hterm fun-denot*) (*G*::*hterm pred-denot*) . $\neg$*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 fun-denot*) (*G*::*'u pred-denot*). $\neg$*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*::*nat fun-denot*) (*G*::*nat pred-denot*) . $\neg$*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.