The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

March 17, 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*
  using *assms* **unfolding** *ground-fatoms-def* **by** (*induction l*) *auto*

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 −
  from *inj all-tree* **have** $\forall\, n.\ n = undiago\ (diago\ n) \wedge (diago\ n) \in S$ **by** *auto*
  then have $\forall\, n.\ \exists\, ds.\ n = undiago\ ds \wedge ds \in S$ **by** *auto*
  then have *undiago* ‘ $S = (UNIV :: nat\ set)$ **by** *auto*
  then show $\neg finite\ S$ **by** (*metis finite-imageI infinite-UNIV-nat*)
qed

lemma *inv-into-f-f*:
  assumes *bij-betw f A B*
  assumes $a \in A$
  shows $(inv\text{-}into\ A\ f)\ (f\ a) = a$
using *assms bij-betw-inv-into-left* **by** *metis*

lemma *f-inv-into-f*:
  assumes *bij-betw f A B*
  assumes $b \in B$
  shows $f\ ((inv\text{-}into\ A\ f)\ b) = b$
using *assms bij-betw-inv-into-right* **by** *metis*

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*
  by (*induction t*) (*simp add*: *map-idI*)

lemma *hterms-of-fterms-fterms-of-hterms*[*simp*]: *hterms-of-fterms* (*fterms-of-hterms ts*) = *ts*
  unfolding *hterms-of-fterms-def fterms-of-hterms-def* **by** (*simp add*: *map-idI*)

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

assumes $ground_t\ t$
shows *fterm-of-hterm* (*hterm-of-fterm* t) = t
using *assms* **by** (*induction* t) (*auto simp add*: *map-idI*)

lemma *fterms-of-hterms-hterms-of-fterms*[*simp*]:
assumes $ground_{ts}\ ts$
shows *fterms-of-hterms* (*hterms-of-fterms ts*) = *ts*
using *assms* **unfolding** *fterms-of-hterms-def hterms-of-fterms-def* **by** (*simp add*: *map-idI*)

lemma *ground-fterm-of-hterm*: $ground_t$ (*fterm-of-hterm* t)
by (*induction* t) (*auto simp add*: *map-idI*)

lemma *ground-fterms-of-hterms*: $ground_{ts}$ (*fterms-of-hterms ts*)
unfolding *fterms-of-hterms-def* **using** *ground-fterm-of-hterm* **by** *auto*

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*)
by (*induction l*) (*simp add*: *ground-fterms-of-hterms*)+

theorem *hlit-of-flit-flit-of-hlit* [*simp*]: *hlit-of-flit* (*flit-of-hlit l*) = *l* **by** (*cases l*) *auto*

theorem *flit-of-hlit-hlit-of-flit* [*simp*]:
assumes $ground_l\ l$
shows *flit-of-hlit* (*hlit-of-flit l*) = *l*
using *assms* **by** (*cases l*) *auto*

lemma *sign-flit-of-hlit*: *sign* (*flit-of-hlit l*) = *sign l* **by** (*cases l*) *auto*

lemma *hlit-of-flit-bij*: *bij-betw hlit-of-flit* {*l*. $ground_l\ l$} *UNIV*
unfolding *bij-betw-def*
proof
show *inj-on hlit-of-flit* {*l*. $ground_l\ l$} **using** *inj-on-inverseI flit-of-hlit-hlit-of-flit*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
next
have ∀ *l*. ∃ *l'*. $ground_l\ l' \wedge l$ = *hlit-of-flit l'*
using *ground-flit-of-hlit hlit-of-flit-flit-of-hlit* **by** *metis*
then show *hlit-of-flit* ‘ {*l*. $ground_l\ l$} = *UNIV* **by** *auto*
qed

lemma *flit-of-hlit-bij*: *bij-betw flit-of-hlit UNIV* {*l*. $ground_l$ *l*}
unfolding *bij-betw-def inj-on-def*
proof
show ∀ *x*∈*UNIV*. ∀ *y*∈*UNIV*. *flit-of-hlit x* = *flit-of-hlit y* ⟶ *x* = *y*
using *ground-flit-of-hlit hlit-of-flit-flit-of-hlit* **by** *metis*
next
have ∀ *l*. $ground_l$ *l* ⟶ (*l* = *flit-of-hlit* (*hlit-of-flit l*)) **using** *hlit-of-flit-flit-of-hlit*
by *auto*
then have {*l*. $ground_l$ *l*} ⊆ *flit-of-hlit* ‘ *UNIV* **by** *blast*
moreover
have ∀ *l*. $ground_l$ (*flit-of-hlit l*) **using** *ground-flit-of-hlit* **by** *auto*
ultimately show *flit-of-hlit* ‘ *UNIV* = {*l*. $ground_l$ *l*} **using** *hlit-of-flit-flit-of-hlit*
ground-flit-of-hlit **by** *auto*
qed

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*))
by (*induction a*) (*simp add*: *ground-fterms-of-hterms*)+

theorem *hatom-of-fatom-fatom-of-hatom* [*simp*]: *hatom-of-fatom* (*fatom-of-hatom*
l) = *l*
by (*cases l*) *auto*

theorem *fatom-of-hatom-hatom-of-fatom* [*simp*]:
assumes $ground_{ts}$ (*snd l*)
shows *fatom-of-hatom* (*hatom-of-fatom l*) = *l*
using *assms* **by** (*cases l*) *auto*

lemma *hatom-of-fatom-bij*: *bij-betw hatom-of-fatom ground-fatoms UNIV*
unfolding *bij-betw-def*
proof
show *inj-on hatom-of-fatom ground-fatoms* **using** *inj-on-inverseI fatom-of-hatom-hatom-of-fatom*
unfolding *ground-fatoms-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
next
have ∀ *a*. ∃ *a*′. $ground_{ts}$ (*snd a*′) ∧ *a* = *hatom-of-fatom a*′
using *ground-fatom-of-hatom hatom-of-fatom-fatom-of-hatom* **by** *metis*
then show *hatom-of-fatom* ‘ *ground-fatoms* = *UNIV* **unfolding** *ground-fatoms-def*
by *blast*
qed

lemma *fatom-of-hatom-bij*: *bij-betw fatom-of-hatom UNIV ground-fatoms*
unfolding *bij-betw-def inj-on-def*
proof
show $\forall x{\in}UNIV.\ \forall y{\in}UNIV.$ *fatom-of-hatom* $x =$ *fatom-of-hatom* $y \longrightarrow x = y$
using *ground-fatom-of-hatom hatom-of-fatom-fatom-of-hatom* **by** *metis*
next
have $\forall a.$ *ground*$_{ts}$ (*snd* a) $\longrightarrow$ (a = *fatom-of-hatom* (*hatom-of-fatom* a)) **using** *hatom-of-fatom-fatom-of-hatom* **by** *auto*
then have *ground-fatoms* $\subseteq$ *fatom-of-hatom* ‘ *UNIV* **unfolding** *ground-fatoms-def* **by** *blast*
moreover
have $\forall l.$ *ground*$_{ts}$ (*snd* (*fatom-of-hatom* l)) **using** *ground-fatom-of-hatom* **by** *auto*
ultimately show *fatom-of-hatom* ‘ *UNIV* = *ground-fatoms*
using *hatom-of-fatom-fatom-of-hatom ground-fatom-of-hatom* **unfolding** *ground-fatoms-def* **by** *auto*
qed

1.4 Enumerations

1.4.1 Enumerating Strings

definition *nat-of-string*:: *string* $\Rightarrow$ *nat* **where**
nat-of-string $\equiv$ (*SOME f*. *bij f*)

definition *string-of-nat*:: *nat* $\Rightarrow$ *string* **where**
string-of-nat $\equiv$ *inv nat-of-string*

lemma *nat-of-string-bij*: *bij nat-of-string*
proof −
have *countable* (*UNIV*::*string set*) **by** *auto*
moreover
have *infinite* (*UNIV*::*string set*) **using** *infinite-UNIV-listI* **by** *auto*
ultimately
obtain x **where** *bij* (x:: *string* $\Rightarrow$ *nat*) **using** *countableE-infinite*[*of UNIV*] **by** *blast*
then show *?thesis* **unfolding** *nat-of-string-def* **using** *someI* **by** *metis*
qed

lemma *string-of-nat-bij*: *bij string-of-nat* **unfolding** *string-of-nat-def* **using** *nat-of-string-bij bij-betw-inv-into* **by** *auto*

lemma *nat-of-string-string-of-nat*[*simp*]: *nat-of-string* (*string-of-nat* n) = n
unfolding *string-of-nat-def*
using *nat-of-string-bij f-inv-into-f*[*of nat-of-string*] **by** *simp*

lemma *string-of-nat-nat-of-string*[*simp*]: *string-of-nat* (*nat-of-string* n) = n
unfolding *string-of-nat-def*
using *nat-of-string-bij inv-into-f-f*[*of nat-of-string*] **by** *simp*

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 by *countable-datatype*
end

lemma *infinite-hatoms*: *infinite* (*UNIV* :: ('*t atom*) *set*)
proof −
let *?diago* = λ*n*. (*string-of-nat n*,[])
let *?undiago* = λ*a*. *nat-of-string* (*fst a*)
have ∀ *n*. *?undiago* (*?diago n*) = *n* **using** *nat-of-string-string-of-nat* **by** *auto*
moreover
have ∀ *n*. *?diago n* ∈ *UNIV* **by** *auto*
ultimately show *infinite* (*UNIV* :: ('*t atom*) *set*) **using** *infinity*[*of ?undiago ?diago UNIV*] **by** *simp*
qed

lemma *nat-of-hatom-bij*: *bij nat-of-hatom*
proof −
let *?S* = *UNIV* :: (('*t*::*countable*) *atom*) *set*
have *countable ?S* **by** *auto*
moreover
have *infinite ?S* **using** *infinite-hatoms* **by** *auto*
ultimately
obtain *x* **where** *bij* (*x* :: *hterm atom* ⇒ *nat*) **using** *countableE-infinite*[*of ?S*] **by** *blast*
then have *bij nat-of-hatom* **unfolding** *nat-of-hatom-def* **using** *someI* **by** *metis*
then show *?thesis* **unfolding** *bij-betw-def inj-on-def* **unfolding** *nat-of-hatom-def* **by** *simp*
qed

lemma *hatom-of-nat-bij*: *bij hatom-of-nat* **unfolding** *hatom-of-nat-def* **using** *nat-of-hatom-bij bij-betw-inv-into* **by** *auto*

lemma *nat-of-hatom-hatom-of-nat*[*simp*]: *nat-of-hatom* (*hatom-of-nat n*) = *n*
unfolding *hatom-of-nat-def*
using *nat-of-hatom-bij f-inv-into-f*[*of nat-of-hatom*] **by** *simp*

lemma *hatom-of-nat-nat-of-hatom*[*simp*]: *hatom-of-nat* (*nat-of-hatom l*) = *l*
unfolding *hatom-of-nat-def*
using *nat-of-hatom-bij inv-into-f-f*[*of nat-of-hatom - UNIV*] **by** *simp*

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*)
using *assms* **unfolding** *fatom-of-nat-def nat-of-fatom-def* **by** *auto*

theorem *undiag-diag-fatom*[*simp*]: *nat-of-fatom* (*fatom-of-nat n*) = *n* **unfolding** *fatom-of-nat-def nat-of-fatom-def* **by** *auto*

lemma *fatom-of-nat-bij*: *bij-betw fatom-of-nat UNIV ground-fatoms*
using *hatom-of-nat-bij bij-betw-trans fatom-of-hatom-bij hatom-of-nat-bij* **unfolding** *fatom-of-nat-def comp-def* **by** *blast*

lemma *ground-fatom-of-nat*: $ground_{ts}$ (*snd* (*fatom-of-nat x*)) **unfolding** *fatom-of-nat-def* **using** *ground-fatom-of-hatom* **by** *auto*

lemma *nat-of-fatom-bij*: *bij-betw nat-of-fatom ground-fatoms UNIV*
using *nat-of-hatom-bij bij-betw-trans hatom-of-fatom-bij hatom-of-nat-bij* **unfolding** *nat-of-fatom-def comp-def* **by** *blast*

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*
using *assms* **by** (*cases T*) *auto*

lemma *treesize-Branching*:
assumes *treesize T* = *Suc n*
shows ∃ *l r*. *T* = *Branching l r*
using *assms* **by** (*cases T*) *auto*

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* = []
by (*induction p*) *auto*

lemma *path-inv-Cons*: *path* (*a*#*ds*) *T* ⟶ (∃ *l r*. *T*=*Branching l r*)
by (*cases T*) (*auto simp add*: *path-inv-Leaf*)

lemma *path-inv-Branching-Left*: *path* (*Left*#*p*) (*Branching l r*) ⟷ *path p l*
using *Left-def Right-def path.cases* **by** (*induction p*) *auto*

lemma *path-inv-Branching-Right*: *path* (*Right*#*p*) (*Branching l r*) ⟷ *path p r*
using *Left-def Right-def path.cases* **by** (*induction p*) *auto*

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
assume *?L* **then show** *?R* **by** (*induction p*) *auto*
next
assume *r*: *?R*
then show *?L*
proof
assume *p* = [] **then show** *?L* **by** *auto*
next
assume ∃ *a p*′. *p*=*a*#*p*′∧ (*a* ⟶ *path p*′ *l*) ∧ (¬*a* ⟶ *path p*′ *r*)
then obtain *a p*′ **where** *p*=*a*#*p*′∧ (*a* ⟶ *path p*′ *l*) ∧ (¬*a* ⟶ *path p*′ *r*)
by *auto*
then show *?L* **by** (*cases a*) *auto*
qed

qed

lemma *path-prefix*:
assumes *path* (*ds1*@*ds2*) *T*
shows *path ds1 T*
using *assms* **proof** (*induction ds1 arbitrary*: *T*)
case (*Cons a ds1*)
then have ∃ *l r*. *T* = *Branching l r* **using** *path-inv-Leaf* **by** (*cases T*) *auto*
then obtain *l r* **where** *p-lr*: *T* = *Branching l r* **by** *auto*
show *?case*
proof (*cases a*)
assume *atrue*: *a*
then have *path* ((*ds1*) @ *ds2*) *l* **using** *p-lr Cons*(*2*) *path-inv-Branching* **by** *auto*
then have *path ds1 l* **using** *Cons*(*1*) **by** *auto*
then show *path* (*a* # *ds1*) *T* **using** *p-lr atrue* **by** *auto*
next
assume *afalse*: ¬*a*
then have *path* ((*ds1*) @ *ds2*) *r* **using** *p-lr Cons*(*2*) *path-inv-Branching* **by** *auto*
then have *path ds1 r* **using** *Cons*(*1*) **by** *auto*
then show *path* (*a* # *ds1*) *T* **using** *p-lr afalse* **by** *auto*
qed
next
case (*Nil*) **then show** *?case* **by** *auto*
qed

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 (*induction T*)
case (*Leaf*)
have *branch* [] *Leaf* **by** *auto*
then show *?case* **by** *blast*
next
case (*Branching* T_1 T_2)
then obtain *b* **where** *branch b* T_1 **by** *auto*
then have *branch* (*Left*#*b*) (*Branching* T_1 T_2) **by** *auto*
then show *?case* **by** *blast*
qed

lemma *branch-inv-Leaf*: *branch b Leaf* ⟷ *b* = []
by (*cases b*) *auto*

lemma *branch-inv-Branching-Left*:
branch (*Left*#*b*) (*Branching l r*) ⟷ *branch b l*
by *auto*

lemma *branch-inv-Branching-Right*:
branch (*Right*#*b*) (*Branching l r*) ⟷ *branch b r*
by *auto*

lemma *branch-inv-Branching*:
branch b (*Branching l r*) ⟷
(∃ *a b*′. *b*=*a*#*b*′∧ (*a* ⟶ *branch b*′ *l*) ∧ (¬*a* ⟶ *branch b*′ *r*))
by (*induction b*) *auto*

lemma *branch-inv-Leaf2*:
T = *Leaf* ⟷ (∀ *b*. *branch b T* ⟶ *b* = [])
proof −
{
assume *T*=*Leaf*
then have ∀ *b*. *branch b T* ⟶ *b* = [] **using** *branch-inv-Leaf* **by** *auto*
}
moreover
{
assume ∀ *b*. *branch b T* ⟶ *b* = []
then have ∀ *b*. *branch b T* ⟶ ¬(∃ *a b*′. *b* = *a* # *b*′) **by** *auto*
then have ∀ *b*. *branch b T* ⟶ ¬(∃ *l r*. *branch b* (*Branching l r*))
using *branch-inv-Branching* **by** *auto*
then have *T*=*Leaf* **using** *has-branch*[*of T*] **by** (*metis branch.elims*(*2*))
}
ultimately show *T* = *Leaf* ⟷ (∀ *b*. *branch b T* ⟶ *b* = []) **by** *auto*
qed

lemma *branch-is-path*:
assumes*branch ds T*
shows *path ds T*
using *assms* **proof** (*induction T arbitrary*: *ds*)
case *Leaf*
then have *ds* = [] **using** *branch-inv-Leaf* **by** *auto*
then show *?case* **by** *auto*
next
case (*Branching* T_1 T_2)
then obtain *a b* **where** *ds-p*: *ds* = *a* # *b* ∧ (*a* ⟶ *branch b* T_1) ∧ (¬ *a* ⟶ *branch b* T_2) **using** *branch-inv-Branching*[*of ds*] **by** *blast*
then have (*a* ⟶ *path b* T_1) ∧ (¬*a* ⟶ *path b* T_2) **using** *Branching* **by** *auto*
then show *?case* **using** *ds-p* **by** (*cases a*) *auto*
qed

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

```
using assms proof (induction T arbitrary: T1 T2)
  case Leaf then show ?case by auto
next
  case (Branching T1' T2')
  {
    assume T1'=Leaf ∧ T2'=Leaf
     then have branch ([] @ [True]) (Branching T1' T2') ∧ branch ([] @ [False])
(Branching T1' T2') by auto
    then have ?case by metis
  }
  moreover
  {
    fix T11 T12
    assume T1' = Branching T11 T12
    then obtain B where branch (B @ [True]) T1'
                      ∧ branch (B @ [False]) T1' using Branching by blast
    then have branch (([True] @ B) @ [True]) (Branching T1' T2')
           ∧ branch (([True] @ B) @ [False]) (Branching T1' T2') by auto
    then have ?case by blast
  }
  moreover
  {
    fix T11 T12
    assume T2' = Branching T11 T12
    then obtain B where branch (B @ [True]) T2'
                      ∧ branch (B @ [False]) T2' using Branching by blast
    then have branch (([False] @ B) @ [True]) (Branching T1' T2')
           ∧ branch (([False] @ B) @ [False]) (Branching T1' T2') by auto
    then have ?case by blast
  }
  ultimately show ?case using tree.exhaust by blast
qed
```

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 using internal.simps by blast

lemma internal-inv-Branching-Left:
  internal (Left#b) (Branching l r) ⟷ internal b l by auto

lemma internal-inv-Branching-Right:
  internal (Right#b) (Branching l r) ⟷ internal b r
by auto
```

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
assume *?L* **then show** *?R* **by** (*metis internal.simps*(*2*) *neq-Nil-conv*)
next
assume *r*: *?R*
then show *?L*
proof
assume *p* = [] **then show** *?L* **by** *auto*
next
assume ∃ *a p'*. *p*=*a*#*p'*∧ (*a* ⟶ *internal p' l*) ∧ (¬*a* ⟶ *internal p' r*)
then obtain *a p'* **where** *p*=*a*#*p'*∧ (*a* ⟶ *internal p' l*) ∧ (¬*a* ⟶ *internal p' r*) **by** *auto*
then show *?L* **by** (*cases a*) *auto*
qed
qed

lemma *internal-is-path*:
assumes *internal ds T*
shows *path ds T*
using *assms* **proof** (*induction T arbitrary*: *ds*)
case *Leaf*
then have *False* **using** *internal-inv-Leaf* **by** *auto*
then show *?case* **by** *auto*
next
case (*Branching* T_1 T_2)
then obtain *a b* **where** *ds-p*: *ds*=[] ∨ *ds* = *a* # *b* ∧ (*a* ⟶ *internal b* T_1) ∧ (¬*a* ⟶ *internal b* T_2) **using** *internal-inv-Branching* **by** *blast*
then have *ds* = [] ∨ (*a* ⟶ *path b* T_1) ∧ (¬*a* ⟶ *path b* T_2) **using** *Branching* **by** *auto*
then show *?case* **using** *ds-p* **by** (*cases a*) *auto*
qed

lemma *internal-prefix*:
assumes *internal* (*ds1*@*ds2*@[*d*]) *T*
shows *internal ds1 T*
using *assms* **proof** (*induction ds1 arbitrary*: *T*)
case (*Cons a ds1*)
then have ∃ *l r*. *T* = *Branching l r* **using** *internal-inv-Leaf* **by** (*cases T*) *auto*
then obtain *l r* **where** *p-lr*: *T* = *Branching l r* **by** *auto*
show *?case*
proof (*cases a*)
assume *atrue*: *a*
then have *internal* ((*ds1*) @ *ds2* @[*d*]) *l* **using** *p-lr Cons*(*2*) *internal-inv-Branching* **by** *auto*
then have *internal ds1 l* **using** *Cons*(*1*) **by** *auto*
then show *internal* (*a* # *ds1*) *T* **using** *p-lr atrue* **by** *auto*
next

```
    assume afalse: ~a
   then have internal ((ds1) @ ds2 @[d]) r using p-lr Cons(2) internal-inv-Branching
by auto
    then have internal ds1 r using Cons(1) by auto
    then show internal (a # ds1) T using p-lr afalse by auto
   qed
next
 case (Nil)
 then have ∃ l r. T = Branching l r using internal-inv-Leaf by (cases T) auto
 then show ?case by auto
qed


lemma internal-branch:
 assumes branch (ds1@ds2@[d]) T
 shows internal ds1 T
using assms proof (induction ds1 arbitrary: T)
 case (Cons a ds1)
 then have ∃ l r. T = Branching l r using branch-inv-Leaf by (cases T) auto
 then obtain l r where p-lr: T = Branching l r by auto
 show ?case
   proof (cases a)
     assume atrue: a
    then have branch (ds1 @ ds2 @ [d]) l using p-lr Cons(2) branch-inv-Branching
by auto
     then have internal ds1 l using Cons(1) by auto
     then show internal (a # ds1) T using p-lr atrue by auto
   next
     assume afalse: ~a
    then have branch ((ds1) @ ds2 @[d]) r using p-lr Cons(2) branch-inv-Branching
by auto
     then have internal ds1 r using Cons(1) by auto
     then show internal (a # ds1) T using p-lr afalse by auto
   qed
next
 case (Nil)
 then have ∃ l r. T = Branching l r using branch-inv-Leaf by (cases T) auto
 then show ?case by auto
qed


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 T1 T2) = Branching (delete ds T1) T2
```

```
| delete (False#ds) (Branching T1 T2) = Branching T1 (delete ds T2)
| delete (a#ds) Leaf = Leaf

lemma delete-Leaf: delete T Leaf = Leaf by (cases T) auto

lemma path-delete:
  assumes path p (delete ds T)
  shows path p T
using assms proof (induction p arbitrary: T ds)
  case Nil
  then show ?case by simp
next
  case (Cons a p)
  then obtain b ds' where bds'-p: ds=b#ds' by (cases ds) auto

  have ∃ dT1 dT2. delete ds T = Branching dT1 dT2 using Cons path-inv-Cons
by auto
  then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto

  then have ∃ T1 T2. T=Branching T1 T2
       by (cases T; cases ds) auto
  then obtain T1 T2 where T1T2-p: T=Branching T1 T2 by auto

  {
    assume a-p: a
    assume b-p: ¬b
    have path (a # p) (delete ds T) using Cons by −
     then have path (a # p) (Branching (T1) (delete ds' T2)) using b-p bds'-p
T1T2-p by auto
    then have path p T1 using a-p by auto
    then have ?case using T1T2-p a-p by auto
  }
  moreover
  {
    assume a-p: ¬a
    assume b-p: b
    have path (a # p) (delete ds T) using Cons by −
     then have path (a # p) (Branching (delete ds' T1) T2) using b-p bds'-p
T1T2-p by auto
    then have path p T2 using a-p by auto
    then have ?case using T1T2-p a-p by auto
  }
  moreover
  {
    assume a-p: a
    assume b-p: b
    have path (a # p) (delete ds T) using Cons by −
     then have path (a # p) (Branching (delete ds' T1) T2) using b-p bds'-p
T1T2-p by auto
```

```
  then have path p (delete ds' T1) using a-p by auto
  then have path p T1 using Cons by auto
  then have ?case using T1T2-p a-p by auto
 }
 moreover
 {
  assume a-p: ¬a
  assume b-p: ¬b
  have path (a # p) (delete ds T) using Cons by −
   then have path (a # p) (Branching T1 (delete ds' T2)) using b-p bds'-p
T1T2-p by auto
  then have path p (delete ds' T2) using a-p by auto
  then have path p T2 using Cons by auto
  then have ?case using T1T2-p a-p by auto
 }
 ultimately show ?case by blast
qed

lemma branch-delete:
 assumes branch p (delete ds T)
 shows branch p T ∨ p=ds
using assms proof (induction p arbitrary: T ds)
 case Nil
 then have delete ds T = Leaf by (cases delete ds T) auto
 then have ds = [] ∨ T = Leaf using delete.elims by blast
 then show ?case by auto
next
 case (Cons a p)
 then obtain b ds' where bds'-p: ds=b#ds' by (cases ds) auto

 have ∃ dT1 dT2. delete ds T = Branching dT1 dT2 using Cons path-inv-Cons
branch-is-path by blast
 then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto

 then have ∃ T1 T2. T=Branching T1 T2
      by (cases T; cases ds) auto
 then obtain T1 T2 where T1T2-p: T=Branching T1 T2 by auto

 {
  assume a-p: a
  assume b-p: ¬b
  have branch (a # p) (delete ds T) using Cons by −
  then have branch (a # p) (Branching (T1) (delete ds' T2)) using b-p bds'-p
T1T2-p by auto
  then have branch p T1 using a-p by auto
  then have ?case using T1T2-p a-p by auto
 }
 moreover
 {
```

```
    assume a-p: ¬a
    assume b-p: b
    have branch (a # p) (delete ds T) using Cons by −
     then have branch (a # p) (Branching (delete ds' T1) T2) using b-p bds'-p
T1T2-p by auto
    then have branch p T2 using a-p by auto
    then have ?case using T1T2-p a-p by auto
  }
  moreover
  {
    assume a-p: a
    assume b-p: b
    have branch (a # p) (delete ds T) using Cons by −
     then have branch (a # p) (Branching (delete ds' T1) T2) using b-p bds'-p
T1T2-p by auto
    then have branch p (delete ds' T1) using a-p by auto
    then have branch p T1 ∨ p = ds' using Cons by metis
    then have ?case using T1T2-p a-p using bds'-p a-p b-p by auto
  }
  moreover
  {
    assume a-p: ¬a
    assume b-p: ¬b
    have branch (a # p) (delete ds T) using Cons by −
     then have branch (a # p) (Branching T1 (delete ds' T2)) using b-p bds'-p
T1T2-p by auto
    then have branch p (delete ds' T2) using a-p by auto
    then have branch p T2 ∨ p = ds' using Cons by metis
    then have ?case using T1T2-p a-p using bds'-p a-p b-p by auto
  }
  ultimately show ?case by blast
qed


lemma branch-delete-postfix:
  assumes path p (delete ds T)
  shows ¬(∃ c cs. p = ds @ c#cs)
using assms proof (induction p arbitrary: T ds)
  case Nil then show ?case by simp
next
  case (Cons a p)
  then obtain b ds' where bds'-p: ds=b#ds' by (cases ds) auto

  have ∃ dT1 dT2. delete ds T = Branching dT1 dT2 using Cons path-inv-Cons
by auto
  then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto

  then have ∃ T1 T2. T=Branching T1 T2
       by (cases T; cases ds) auto
```

```
 then obtain T1 T2 where T1T2-p: T=Branching T1 T2 by auto

 {
  assume a-p: a
  assume b-p: ¬b
  then have ?case using T1T2-p a-p b-p bds'-p by auto
 }
 moreover
 {
  assume a-p: ¬a
  assume b-p: b
  then have ?case using T1T2-p a-p b-p bds'-p by auto
 }
 moreover
 {
  assume a-p: a
  assume b-p: b
  have path (a # p) (delete ds T) using Cons by −
   then have path (a # p) (Branching (delete ds' T1) T2) using b-p bds'-p
T1T2-p by auto
  then have path p (delete ds' T1) using a-p by auto
  then have ¬ (∃ c cs. p = ds' @ c # cs) using Cons by auto
  then have ?case using T1T2-p a-p b-p bds'-p by auto
 }
 moreover
 {
  assume a-p: ¬a
  assume b-p: ¬b
  have path (a # p) (delete ds T) using Cons by −
   then have path (a # p) (Branching T1 (delete ds' T2)) using b-p bds'-p
T1T2-p by auto
  then have path p (delete ds' T2) using a-p by auto
  then have ¬ (∃ c cs. p = ds' @ c # cs) using Cons by auto
  then have ?case using T1T2-p a-p b-p bds'-p by auto
 }
 ultimately show ?case by blast
qed

lemma treezise-delete:
 assumes internal p T
 shows treesize (delete p T) < treesize T
using assms proof (induction p arbitrary: T)
 case (Nil)
 then have ∃ T1 T2. T = Branching T1 T2 by (cases T) auto
 then obtain T1 T2 where T1T2-p: T = Branching T1 T2 by auto
 then show ?case by auto
next
 case (Cons a p)
 then have ∃ T1 T2. T = Branching T1 T2 using path-inv-Cons internal-is-path
```

```
by blast
  then obtain T1 T2 where T1T2-p: T = Branching T1 T2 by auto
  show ?case
    proof (cases a)
      assume a-p: a
     from a-p have delete (a#p) T = (Branching (delete p T1) T2) using T1T2-p
by auto
      moreover
      from a-p have internal p T1 using T1T2-p Cons by auto
      then have treesize (delete p T1) < treesize T1 using Cons by auto
      ultimately
      show ?thesis using T1T2-p by auto
    next
      assume a-p: ¬a
     from a-p have delete (a#p) T = (Branching T1 (delete p T2)) using T1T2-p
by auto
      moreover
      from a-p have internal p T2 using T1T2-p Cons by auto
      then have treesize (delete p T2) < treesize T2 using Cons by auto
      ultimately
      show ?thesis using T1T2-p by auto
    qed
qed


fun cutoff :: (dir list ⇒ bool) ⇒ dir list ⇒ tree ⇒ 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 treesize-cutoff: treesize (cutoff red ds T) ≤ treesize T
proof (induction T arbitrary: ds)
  case Leaf then show ?case by auto
next
  case (Branching T1 T2)
  then have treesize (cutoff red (ds@[Left]) T1) + treesize (cutoff red (ds@[Right])
T2) ≤ treesize T1 + treesize T2 using add-mono by blast
  then show ?case by auto
qed

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*))
using *assms* **proof** (*induction T arbitrary*: *ds*)
case (*Leaf*)
let *?T* = *cutoff red ds Leaf*
{
fix *b*
assume *branch b ?T*
then have *branch b Leaf* **by** *auto*
then have *red*(*ds*@*b*) **using** *Leaf* **by** *auto*
}
then show *?case* **by** *simp*
next
case (*Branching* T_1 T_2)
let *?T* = *cutoff red ds* (*Branching* T_1 T_2)
from *Branching* **have** ∀ *p*. *branch* (*Left*#*p*) (*Branching* T_1 T_2) ⟶ *red* (*ds* @ (*Left*#*p*)) **by** *blast*
then have ∀ *p*. *branch p* T_1 ⟶ *red* (*ds* @ (*Left*#*p*)) **by** *auto*
then have *anybranch* T_1 (λ*p*. *red* ((*ds* @ [*Left*]) @ *p*)) **by** *auto*
then have *aa*: *anybranch* (*cutoff red* (*ds* @ [*Left*]) T_1) (λ*p*. *red* ((*ds* @ [*Left*]) @ *p*))
using *Branching* **by** *blast*

from *Branching* **have** ∀ *p*. *branch* (*Right*#*p*) (*Branching* T_1 T_2) ⟶ *red* (*ds* @ (*Right*#*p*)) **by** *blast*
then have ∀ *p*. *branch p* T_2 ⟶ *red* (*ds* @ (*Right*#*p*)) **by** *auto*
then have *anybranch* T_2 (λ*p*. *red* ((*ds* @ [*Right*]) @ *p*)) **by** *auto*
then have *bb*: *anybranch* (*cutoff red* (*ds* @ [*Right*]) T_2) (λ*p*. *red* ((*ds* @ [*Right*]) @ *p*))
using *Branching* **by** *blast*
{
fix *b*
assume *b-p*: *branch b ?T*
have *red ds* ∨ ¬*red ds* **by** *auto*
then have *red*(*ds*@*b*)
proof
assume *ds-p*: *red ds*
then have *?T* = *Leaf* **by** *auto*
then have *b* = [] **using** *b-p branch-inv-Leaf* **by** *auto*
then show *red*(*ds*@*b*) **using** *ds-p* **by** *auto*
next
assume *ds-p*: ¬*red ds*
let $?T_1' = cutoff\ red\ (ds@[Left])\ T_1$
let $?T_2' = cutoff\ red\ (ds@[Right])\ T_2$

```
      from ds-p have ?T = Branching ?T1' ?T2' by auto
      from this b-p obtain a b' where b = a # b' ∧ (a ⟶ branch b' ?T1') ∧
(¬a ⟶ branch b' ?T2' ) using branch-inv-Branching[of b ?T1' ?T2'] by auto
      then show red(ds@b) using aa bb by (cases a) auto
    qed
  }
  then show ?case by blast
qed

lemma cutoff-branch:
  assumes anybranch T (λp. red p)
  shows anybranch (cutoff red [] T) (λp. red p)
  using assms cutoff-branch'[of T red []] by auto

lemma cutoff-internal':
  assumes anybranch T (λb. red(ds@b))
  shows anyinternal (cutoff red ds T) (λb. ¬red(ds@b))
using assms proof (induction T arbitrary: ds)
  case (Leaf) then show ?case using internal-inv-Leaf by simp
next
  case (Branching T1 T2)
  let ?T = cutoff red ds (Branching T1 T2)
  from Branching have ∀ p. branch (Left#p) (Branching T1 T2) ⟶ red (ds @
(Left#p)) by blast
  then have ∀ p. branch p T1 ⟶ red (ds @ (Left#p)) by auto
  then have anybranch T1 (λp. red ((ds @ [Left]) @ p)) by auto
  then have aa: anyinternal (cutoff red (ds @ [Left]) T1) (λp. ¬ red ((ds @ [Left])
@ p)) using Branching by blast

  from Branching have ∀ p. branch (Right#p) (Branching T1 T2) ⟶ red (ds @
(Right#p)) by blast
  then have ∀ p. branch p T2 ⟶ red (ds @ (Right#p)) by auto
  then have anybranch T2 (λp. red ((ds @ [Right]) @ p)) by auto
  then have bb: anyinternal (cutoff red (ds @ [Right]) T2) (λp. ¬ red ((ds @
[Right]) @ p)) using Branching by blast
  {
    fix p
    assume b-p: internal p ?T
    then have ds-p: ¬red ds using internal-inv-Leaf by auto
    have p=[] ∨ p≠[] by auto
    then have ¬red(ds@p)
      proof
        assume p=[] then show ¬red(ds@p) using ds-p by auto
      next
        let ?T1' = cutoff red (ds@[Left]) T1
        let ?T2' = cutoff red (ds@[Right]) T2
        assume p≠[]
        moreover
        have ?T = Branching ?T1' ?T2' using ds-p by auto
```

ultimately
obtain *a p'* **where** *b-p*: $p = a \# p' \wedge$
$(a \longrightarrow internal\ p'\ (cutoff\ red\ (ds\ @\ [Left])\ T_1)) \wedge$
$(\neg\ a \longrightarrow internal\ p'\ (cutoff\ red\ (ds\ @\ [Right])\ T_2))$
using *b-p internal-inv-Branching*[*of p* $?T_1'$ $?T_2'$] **by** *auto*
then have $\neg red(ds\ @\ [a]\ @\ p')$ **using** *aa bb* **by** (*cases a*) *auto*
then show $\neg red(ds\ @\ p)$ **using** *b-p* **by** *simp*
qed
}
then show *?case* **by** *blast*
qed

lemma *cutoff-internal*:
assumes *anybranch T red*
shows *anyinternal* (*cutoff red* [] *T*) ($\lambda p.\ \neg red\ p$)
using *assms cutoff-internal'*[*of T red* []] **by** *auto*

lemma *cutoff-branch-internal'*:
assumes *anybranch T red*
shows *anyinternal* (*cutoff red* [] *T*) ($\lambda p.\ \neg red\ p$) $\wedge$ *anybranch* (*cutoff red* [] *T*) ($\lambda p.\ red\ p$)
using *assms cutoff-internal*[*of T*] *cutoff-branch*[*of T*] **by** *blast*

lemma *cutoff-branch-internal*:
assumes *anybranch T red*
shows $\exists\ T'.$ *anyinternal T'* ($\lambda p.\ \neg red\ p$) $\wedge$ *anybranch T'* ($\lambda p.\ red\ p$)
using *assms cutoff-branch-internal'* **by** *blast*

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*}
proof (*rule subsetI*; *rule Set.UnCI*)
let *?subtree* = *subtree T*
fix *x*
assume *asm*: *x* $\in$ *?subtree ds*
assume *x* $\notin$ {*ds*}

then have $x \neq ds$ **by** *simp*
then have $\exists\, e\ d.\ x = ds\ @\ [d]\ @\ e$ **using** *asm list.exhaust* **by** *auto*
then have $(\exists\, e.\ x = ds\ @\ [Left]\ @\ e) \vee (\exists\, e.\ x = ds\ @\ [Right]\ @\ e)$ **using** *bool.exhaust* **by** *auto*
then show $x \in$ *?subtree* $(ds\ @\ [Left]) \cup$ *?subtree* $(ds\ @\ [Right])$ **using** *asm* **by** *auto*
qed

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* $(f\ n) = n$
using *assms* **proof** (*induction n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*) **then show** *?case* **by** (*metis length-append-singleton*)
qed

lemma *chain-prefix*:
assumes *wf-infpath f*
assumes $n_1 \leq n_2$
shows $\exists\, a.\ (f\ n_1)\ @\ a = (f\ n_2)$
using *assms* **proof** (*induction* n_2)
case (*Suc* n_2)
then have $n_1 \leq n_2 \vee n_1 = Suc\ n_2$ **by** *auto*
then show *?case*
proof
assume $n_1 \leq n_2$
then obtain *a* **where** *a*: $f\ n_1\ @\ a = f\ n_2$ **using** *Suc* **by** *auto*
have *b*: $\exists\, b.\ f\ (Suc\ n_2) = f\ n_2\ @\ [b]$ **using** *Suc* **by** *auto*
from *a b* **have** $\exists\, b.\ f\ n_1\ @\ (a\ @\ [b]) = f\ (Suc\ n_2)$ **by** *auto*
then show $\exists\, c.\ f\ n_1\ @\ c = f\ (Suc\ n_2)$ **by** *blast*
next
assume $n_1 = Suc\ n_2$
then have $f\ n_1\ @\ [] = f\ (Suc\ n_2)$ **by** *auto*
then show $\exists\, a.\ f\ n_1\ @\ a = f\ (Suc\ n_2)$ **by** *auto*
qed
qed *auto*

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 −
from *chain* $n_1 n_2$ **have** $\exists\, a.\ f\ n_1\ @\ a = f\ n_2$ **using** *chain-prefix* **by** *blast*
then obtain a **where** *a-p*: $f\ n_1\ @\ a = f\ n_2$ **by** *auto*
have $(f\ n_1\ @\ a)\ !\ i = f\ n_1\ !\ i$ **using** *smalli* **by** (*simp add*: *nth-append*)
then show *?thesis* **using** *a-p* **by** *auto*
qed

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 −
let *?subtree* = *subtree T*
{
assume *asms*: *finite*(*?subtree*(*ds* @ [*Left*]))
finite(*?subtree*(*ds* @ [*Right*]))
have *?subtree ds* ⊆ *?subtree* (*ds* @ [*Left*]) ∪ *?subtree* (*ds* @ [*Right*]) ∪ {*ds*}
using *subtree-pos* **by** *auto*
then have *finite*(*?subtree* (*ds*)) **using** *asms* **by** (*simp add*: *finite-subset*)
}
then show ¬*finite*(*?subtree* (*ds* @ [*Left*])) ∨ ¬*finite*(*?subtree* (*ds* @ [*Right*]))
using *inf* **by** *auto*
qed

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
let *?subtree* = *subtree T*
let *?nextnode* = λ*ds*. (*if* ¬*finite* (*?subtree* (*ds* @ [*Left*])) *then ds* @ [*Left*] *else ds* @ [*Right*])

let *?c* = *buildchain ?nextnode*

have *is-chain*: *wf-infpath ?c* **by** *auto*

from *wellformed* **have** *prefix*: ∀ *ds d*. (*ds* @ *d*) ∈ *T* ⟶ *ds* ∈ *T* **by** *blast*

{
fix *n*
have (*?c n*) ∈ *T* ∧ ¬*finite* (*?subtree* (*?c n*))
proof (*induction n*)

```
    case 0
    have ∃ ds. ds ∈ T using inf by (simp add: not-finite-existsD)
    then obtain ds where ds ∈ T by auto
    then have ([]@ds) ∈ T by auto
    then have [] ∈ T using prefix by blast
    then show ?case using inf by auto
  next
    case (Suc n)
    from Suc have next-in:  (?c n) ∈ T by auto
    from Suc have next-inf: ¬finite (?subtree (?c n)) by auto

    from next-inf have next-next-inf:
      ¬finite (?subtree (?nextnode (?c n)))
        using inf-subs by auto
    then have ∃ ds. ds ∈ ?subtree (?nextnode (?c n))
     by (simp add: not-finite-existsD)
    then obtain ds where dss: ds ∈ ?subtree (?nextnode (?c n)) by auto
    then have ds ∈ T ∃ suf. ds = (?nextnode (?c n)) @ suf by auto
    then obtain suf where ds ∈ T ∧ ds = (?nextnode (?c n)) @ suf by auto
    then have (?nextnode (?c n)) ∈ T
     using prefix by blast

    then have (?c (Suc n)) ∈ T by auto
    then show ?case using next-next-inf by auto
  qed
 }
 then show wf-infpath ?c ∧ (∀ n. (?c n)∈ T)  using is-chain by auto
qed

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: (lᶜ)ᶜ = l by (cases l) auto

lemma cancel-comp2:
  assumes asm: l₁ᶜ = l₂ᶜ
  shows l₁ = l₂
proof −
  from asm have (l₁ᶜ)ᶜ = (l₂ᶜ)ᶜ by auto
  then have l₁ = (l₂ᶜ)ᶜ using cancel-comp1[of l₁] by auto
  then show ?thesis using cancel-comp1[of l₂] by auto
qed
```

lemma *comp-exi1*: $\exists\, l'.\ l' = l^c$ **by** (*cases l*) *auto*

lemma *comp-exi2*: $\exists\, l.\ l' = l^c$
proof
  show $l' = (l'^c)^c$ **using** *cancel-comp1*[*of l'*] **by** *auto*
qed

lemma *comp-swap*: ${l_1}^c = l_2 \longleftrightarrow l_1 = {l_2}^c$
proof −
  have ${l_1}^c = l_2 \longrightarrow l_1 = {l_2}^c$ **using** *cancel-comp1*[*of* l_1] **by** *auto*
  moreover
  have $l_1 = {l_2}^c \longrightarrow {l_1}^c = l_2$ **using** *cancel-comp1* **by** *auto*
  ultimately
  show *?thesis* **by** *auto*
qed

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* $l_2 \longleftrightarrow l_2 = {l_1}^c$
by (*cases* l_1; *cases* l_2) *auto*

lemma *sign-comp-atom*: *sign* $l_1 \neq$ *sign* $l_2 \wedge$ *get-atom* $l_1 =$ *get-atom* $l_2 \longleftrightarrow l_2 = {l_1}^c$
by (*cases* l_1; *cases* l_2) *auto*

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*: $(L^C)^C = L$
apply (*auto simp add*: *cancel-comp1*)
apply (*metis imageI cancel-comp1*)
done

lemma *cancel-compls2*:
  assumes *asm*: ${L_1}^C = {L_2}^C$
  shows $L_1 = L_2$
proof −
  from *asm* **have** $({L_1}^C)^C = ({L_2}^C)^C$ **by** *auto*
  then show *?thesis* **using** *cancel-compls1*[*of* L_1] *cancel-compls1*[*of* L_2] **by** *simp*
qed

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* ∈ *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* ≡ $\bigcup$ *l*∈*L*. $vars_l$ *l*

lemma $ground\text{-}vars_t$:
assumes $ground_t$ *t*
shows $vars_t$ *t* = {}
using *assms* **by** (*induction t*) *auto*

lemma $ground_{ts}\text{-}vars_{ts}$:
assumes $ground_{ts}$ *ts*
shows $vars_{ts}$ *ts* = {}
using *assms* $ground\text{-}vars_t$ **by** *auto*

lemma $ground_l\text{-}vars_l$:
assumes $ground_l$ *l*
shows $vars_l$ *l* = {}
unfolding $vars_l\text{-}def$ **using** *assms* $ground\text{-}vars_t$ **by** *auto*

lemma $ground_{ls}\text{-}vars_{ls}$:
assumes $ground_{ls}$ *L*
shows $vars_{ls}$ *L* = {} **unfolding** $vars_{ls}\text{-}def$ **using** *assms* $ground_l\text{-}vars_l$ **by** *auto*

lemma *ground-comp*: $ground_l$ (l^c) ⟷ $ground_l$ *l* **by** (*cases l*) *auto*

lemma *ground-compls*: $ground_{ls}$ (L^C) ⟷ $ground_{ls}$ *L* **using** *ground-comp* **by** *auto*

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**

$eval_l\ E\ F\ G\ (Pos\ p\ ts) \longleftrightarrow\ G\ p\ (eval_{ts}\ E\ F\ ts)$
| $eval_l\ E\ F\ G\ (Neg\ p\ ts) \longleftrightarrow \neg G\ p\ (eval_{ts}\ E\ F\ ts)$

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

definition $eval_{cs} :: 'u\ fun\text{-}denot \Rightarrow 'u\ pred\text{-}denot \Rightarrow fterm\ clause\ set \Rightarrow bool$ **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 $eval_t\ E\ F\ t = eval_t\ E'\ F\ t$
using *assms* **proof** (*induction t*)
case (*Var x*)
then have *False* **by** *auto*
then show *?case* **by** *auto*
next
case (*Fun f ts*)
then have $\forall t \in set\ ts.\ ground_t\ t$ **by** *auto*
then have $\forall t \in set\ ts.\ eval_t\ E\ F\ t = eval_t\ E'\ F\ t$ **using** *Fun* **by** *auto*
then have $eval_{ts}\ E\ F\ ts = eval_{ts}\ E'\ F\ ts$ **by** *auto*
then have $F\ f\ (map\ (eval_t\ E\ F)\ ts) = F\ f\ (map\ (eval_t\ E'\ F)\ ts)$ **by** *metis*
then show *?case* **by** *simp*
qed

lemma *ground-var-denotts*:
assumes $ground_{ts}\ ts$
shows $eval_{ts}\ E\ F\ ts = eval_{ts}\ E'\ F\ ts$
using *assms ground-var-denott* **by** (*metis map-eq-conv*)

lemma *ground-var-denot*:
assumes $ground_l\ l$
shows $eval_l\ E\ F\ G\ l = eval_l\ E'\ F\ G\ l$
using *assms* **proof** (*induction l*)
case *Pos* **then show** *?case* **using** *ground-var-denotts* **by** (*metis* $eval_l.simps(1)$ *literal.sel*(*3*))
next
case *Neg* **then show** *?case* **using** *ground-var-denotts* **by** (*metis* $eval_l.simps(2)$ *literal.sel*(*4*))
qed

8 Substitutions

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

fun *sub* :: *fterm* ⇒ *substitution* ⇒ *fterm* (**infixl** ‹·$_t$› *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* ⇒ *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*: $L \cdot_{ls} \sigma = \{l \cdot_l \sigma | l.\ l \in L\}$ **by** *auto*

definition *instance-of*$_t$ :: *fterm* ⇒ *fterm* ⇒ *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* ⇒ *fterm list* ⇒ *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* ⇒ *fterm literal* ⇒ *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* ⇒ *fterm clause* ⇒ *bool* **where**
instance-of$_{ls}$ $C_1\ C_2 \longleftrightarrow (\exists \sigma.\ C_1 = C_2 \cdot_{ls} \sigma)$

lemma *comp-sub*: $(l^c) \cdot_l \sigma = (l \cdot_l \sigma)^c$
by (*cases l*) *auto*

lemma *compls-subls*: $(L^C) \cdot_{ls} \sigma = (L \cdot_{ls} \sigma)^C$
using *comp-sub* **apply** *auto*
apply (*metis image-eqI*)
done

lemma *subls-union*: $(L_1 \cup L_2) \cdot_{ls} \sigma = (L_1 \cdot_{ls} \sigma) \cup (L_2 \cdot_{ls} \sigma)$ **by** *auto*

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

8.1 The Empty Substitution

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

lemma *empty-subt*: $(t :: fterm) \cdot_t \varepsilon = t$
by (*induction t*) (*auto simp add*: *map-idI*)

lemma *empty-subts*: $ts \cdot_{ts} \varepsilon = ts$
using *empty-subt* **by** *auto*

lemma *empty-subl*: $l \cdot_l \varepsilon = l$
using *empty-subts* **by** (*cases l*) *auto*

lemma *empty-subls*: $L \cdot_{ls} \varepsilon = L$
using *empty-subl* **by** *auto*

lemma *instance-of$_t$-self*: *instance-of$_t$ t t*
unfolding *instance-of$_t$-def*
proof
  show $t = t \cdot_t \varepsilon$ **using** *empty-subt* **by** *auto*
qed

lemma *instance-of$_{ts}$-self*: *instance-of$_{ts}$ ts ts*
unfolding *instance-of$_{ts}$-def*
proof
  show $ts = ts \cdot_{ts} \varepsilon$ **using** *empty-subts* **by** *auto*
qed

lemma *instance-of$_l$-self*: *instance-of$_l$ l l*
unfolding *instance-of$_l$-def*
proof
  show $l = l \cdot_l \varepsilon$ **using** *empty-subl* **by** *auto*
qed

lemma *instance-of$_{ls}$-self*: *instance-of$_{ls}$ L L*
unfolding *instance-of$_{ls}$-def*
proof
  show $L = L \cdot_{ls} \varepsilon$ **using** *empty-subls* **by** *auto*
qed

8.2 Substitutions and Ground Terms

lemma *ground-sub*:
  assumes *ground$_t$ t*
  shows $t \cdot_t \sigma = t$
using *assms* **by** (*induction t*) (*auto simp add*: *map-idI*)

lemma *ground-subs*:
  assumes *ground$_{ts}$ ts*
  shows $ts \cdot_{ts} \sigma = ts$
using *assms ground-sub* **by** (*simp add*: *map-idI*)

lemma *ground$_l$-subs*:
  assumes *ground$_l$ l*
  shows $l \cdot_l \sigma = l$

using *assms ground-subs* **by** (*cases l*) *auto*

lemma *ground$_{ls}$-subls*:
assumes *ground*: *ground$_{ls}$ L*
shows $L \cdot_{ls} \sigma = L$
proof −
{
fix *l*
assume *l-L*: $l \in L$
then have *ground$_l$ l* **using** *ground* **by** *auto*
then have $l = l \cdot_l \sigma$ **using** *ground$_l$-subs* **by** *auto*
moreover
then have $l \cdot_l \sigma \in L \cdot_{ls} \sigma$ **using** *l-L* **by** *auto*
ultimately
have $l \in L \cdot_{ls} \sigma$ **by** *auto*
}
moreover
{
fix *l*
assume *l-L*: $l \in L \cdot_{ls} \sigma$
then obtain l' **where** *l'-p*: $l' \in L \wedge l' \cdot_l \sigma = l$ **by** *auto*
then have $l' = l$ **using** *ground ground$_l$-subs* **by** *auto*
from *l-L l'-p this* **have** $l \in L$ **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

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 (*induction t*)
case (*Var x*)
have $((Var\ x) \cdot_t \sigma_1) \cdot_t \sigma_2 = (\sigma_1\ x) \cdot_t \sigma_2$ **by** *simp*
also have $... = (\sigma_1 \cdot \sigma_2)\ x$ **unfolding** *composition-def* **by** *simp*
finally show *?case* **by** *auto*
next
case (*Fun t ts*)
then show *?case* **unfolding** *composition-def* **by** *auto*
qed

lemma *composition-conseq2ts*: $(ts \cdot_{ts} \sigma_1) \cdot_{ts} \sigma_2 = ts \cdot_{ts} (\sigma_1 \cdot \sigma_2)$
using *composition-conseq2t* **by** *auto*

lemma *composition-conseq2l*: $(l \cdot_l \sigma_1) \cdot_l \sigma_2 = l \cdot_l (\sigma_1 \cdot \sigma_2)$
using *composition-conseq2t* **by** (*cases l*) *auto*

lemma *composition-conseq2ls*: $(L \cdot_{ls} \sigma_1) \cdot_{ls} \sigma_2 = L \cdot_{ls} (\sigma_1 \cdot \sigma_2)$
using *composition-conseq2l* **apply** *auto*
apply (*metis imageI*)
done

lemma *composition-assoc*: $\sigma_1 \cdot (\sigma_2 \cdot \sigma_3) = (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$
proof
  fix x
  show $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3))\ x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3)\ x$
    by (*simp only*: *composition-def composition-conseq2t*)
qed

lemma *empty-comp1*: $(\sigma \cdot \varepsilon) = \sigma$
proof
  fix x
  show $(\sigma \cdot \varepsilon)\ x = \sigma\ x$ **unfolding** *composition-def* **using** *empty-subt* **by** *auto*
qed

lemma *empty-comp2*: $(\varepsilon \cdot \sigma) = \sigma$
proof
  fix x
  show $(\varepsilon \cdot \sigma)\ x = \sigma\ x$ **unfolding** *composition-def* **by** *simp*
qed

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 −
  from t_{12} **obtain** σ_{12} **where** $t_1 = t_2 \cdot_t \sigma_{12}$
    unfolding *instance-of$_t$-def* **by** *auto*
  moreover
  from t_{23} **obtain** σ_{23} **where** $t_2 = t_3 \cdot_t \sigma_{23}$
    unfolding *instance-of$_t$-def* **by** *auto*
  ultimately
  have $t_1 = (t_3 \cdot_t \sigma_{23}) \cdot_t \sigma_{12}$ **by** *auto*
  then have $t_1 = t_3 \cdot_t (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2t* **by** *simp*
  then show *?thesis* **unfolding** *instance-of$_t$-def* **by** *auto*
qed

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 −
  from ts_{12} **obtain** σ_{12} **where** $ts_1 = ts_2 \cdot_{ts} \sigma_{12}$
    unfolding *instance-of$_{ts}$-def* **by** *auto*

moreover
from ts_{23} **obtain** σ_{23} **where** $ts_2 = ts_3 \cdot_{ts} \sigma_{23}$
unfolding *instance-of*$_{ts}$*-def* **by** *auto*
ultimately
have $ts_1 = (ts_3 \cdot_{ts} \sigma_{23}) \cdot_{ts} \sigma_{12}$ **by** *auto*
then have $ts_1 = ts_3 \cdot_{ts} (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2ts* **by** *simp*
then show *?thesis* **unfolding** *instance-of*$_{ts}$*-def* **by** *auto*
qed

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 −
from l_{12} **obtain** σ_{12} **where** $l_1 = l_2 \cdot_{l} \sigma_{12}$
unfolding *instance-of*$_{l}$*-def* **by** *auto*
moreover
from l_{23} **obtain** σ_{23} **where** $l_2 = l_3 \cdot_{l} \sigma_{23}$
unfolding *instance-of*$_{l}$*-def* **by** *auto*
ultimately
have $l_1 = (l_3 \cdot_{l} \sigma_{23}) \cdot_{l} \sigma_{12}$ **by** *auto*
then have $l_1 = l_3 \cdot_{l} (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2l* **by** *simp*
then show *?thesis* **unfolding** *instance-of*$_{l}$*-def* **by** *auto*
qed

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 −
from L_{12} **obtain** σ_{12} **where** $L_1 = L_2 \cdot_{ls} \sigma_{12}$
unfolding *instance-of*$_{ls}$*-def* **by** *auto*
moreover
from L_{23} **obtain** σ_{23} **where** $L_2 = L_3 \cdot_{ls} \sigma_{23}$
unfolding *instance-of*$_{ls}$*-def* **by** *auto*
ultimately
have $L_1 = (L_3 \cdot_{ls} \sigma_{23}) \cdot_{ls} \sigma_{12}$ **by** *auto*
then have $L_1 = L_3 \cdot_{ls} (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2ls* **by** *simp*
then show *?thesis* **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
qed

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 −
let *?L* = $\{l \in C.\ \exists l' \in L'.\ l \cdot_{l} lmbd = l'\}$

```
  have ?L ⊆ C by auto
  moreover
  have ?L ·_ls lmbd = L'
    proof (rule Orderings.order-antisym; rule Set.subsetI)
      fix l'
      assume l'L: l' ∈ L'
      from inst-C have {l ·_l lmbd|l. l ∈ C} = C' unfolding subls-def2 by −
       then have ∃ l. l' = l ·_l lmbd ∧ l ∈ C ∧ l ·_l lmbd ∈ L' using L'sub l'L by
auto
      then have  l' ∈ {l ∈ C. l ·_l lmbd ∈ L'} ·_ls lmbd by auto
      then show  l' ∈ {l ∈ C. ∃ l'∈L'. l ·_l lmbd = l'} ·_ls lmbd by auto
    qed auto
  moreover
  have (C−?L) ·_ls lmbd = C' − L' using inst-C by auto
  ultimately show ?thesis
    by blast
qed

lemma relevant-vars-subt:
  assumes ∀ x ∈ vars_t t. σ_1 x = σ_2 x
  shows  t ·_t σ_1 = t ·_t σ_2
using assms proof (induction t)
  case (Fun f ts)
  have f: ∀ t. t ∈ set ts ⟶ vars_t t ⊆ vars_ts ts by (induction ts) auto
  have ∀ t∈set ts. t ·_t σ_1 = t ·_t σ_2
    proof
      fix t
      assume tints: t ∈ set ts
      then have ∀ x ∈ vars_t t. σ_1 x = σ_2 x using f Fun(2) by auto
      then show t ·_t σ_1 = t ·_t σ_2 using Fun tints by auto
    qed
  then have ts ·_ts σ_1 = ts ·_ts σ_2 by auto
  then show ?case by auto
qed auto

lemma relevant-vars-subts:
  assumes asm: ∀ x ∈ vars_ts ts. σ_1 x = σ_2 x
  shows ts ·_ts σ_1 = ts ·_ts σ_2
proof −
   have f: ∀ t. t ∈ set ts ⟶ vars_t t ⊆ vars_ts ts by (induction ts) auto
   have ∀ t∈set ts. t ·_t σ_1 = t ·_t σ_2
    proof
      fix t
      assume tints: t ∈ set ts
      then have ∀ x ∈ vars_t t. σ_1 x = σ_2 x using f asm by auto
      then show t ·_t σ_1 = t ·_t σ_2 using relevant-vars-subt tints by auto
    qed
  then show ?thesis by auto
qed
```

lemma *relevant-vars-subl*:
assumes $\forall\, x \in vars_l\ l.\ \sigma_1\ x = \sigma_2\ x$
shows $l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
using *assms* **proof** (*induction l*)
case (*Pos p ts*)
then show *?case* **using** *relevant-vars-subts* **unfolding** $vars_l$*-def* **by** *auto*
next
case (*Neg p ts*)
then show *?case* **using** *relevant-vars-subts* **unfolding** $vars_l$*-def* **by** *auto*
qed

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 −
have *f*: $\forall\, l.\ l \in L \longrightarrow vars_l\ l \subseteq vars_{ls}\ L$ **unfolding** $vars_{ls}$*-def* **by** *auto*
have $\forall\, l \in L.\ l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
proof
fix *l*
assume *linls*: $l \in L$
then have $\forall\, x \in vars_l\ l.\ \sigma_1\ x = \sigma_2\ x$ **using** *f asm* **by** *auto*
then show $l \cdot_l \sigma_1 = l \cdot_l \sigma_2$ **using** *relevant-vars-subl linls* **by** *auto*
qed
then show *?thesis* **by** (*meson image-cong*)
qed

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 −
let $?\eta = \lambda x.\ if\ x \in vars_{ls}\ C\ then\ lmbd\ x\ else\ \mu\ x$
have $\forall\, x \in vars_{ls}\ C.\ ?\eta\ x = lmbd\ x$ **by** *auto*
then have $C \cdot_{ls} ?\eta = C \cdot_{ls} lmbd$ **using** *relevant-vars-subls*[*of C ?η lmbd*] **by** *auto*
then have $C \cdot_{ls} ?\eta = C'$ **using** *CC'* **by** *auto*
moreover
have $\forall\, x \in vars_{ls}\ D.\ ?\eta\ x = \mu\ x$ **using** *dist* **by** *auto*
then have $D \cdot_{ls} ?\eta = D \cdot_{ls} \mu$ **using** *relevant-vars-subls*[*of D ?η μ*] **by** *auto*
then have $D \cdot_{ls} ?\eta = D'$ **using** *DD'* **by** *auto*
ultimately
show *?thesis* **by** *auto*
qed

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 $std_2 :: fterm\ clause \Rightarrow fterm\ clause$ **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 $\exists x'.\ x = y@x'$
using *assms* **by** (*induction t*) *auto*

lemma *std-apart-apart'*:
assumes $x \in vars_l\ (l \cdot_l (\lambda x.\ Var\ \ (y@x)))$
shows $\exists x'.\ x = y@x'$
using *assms* **unfolding** $vars_l$*-def* **using** *std-apart-apart''* **by** (*cases l*) *auto*

lemma *std-apart-apart*: $vars_{ls}\ (std_1\ C_1) \cap vars_{ls}\ (std_2\ C_2) = \{\}$
proof −
{
fix *x*
assume *xin*: $x \in vars_{ls}\ (std_1\ C_1) \cap vars_{ls}\ (std_2\ C_2)$
from *xin* **have** $x \in vars_{ls}\ (std_1\ C_1)$ **by** *auto*
then have $\exists x'.\ x{=}''1''\ @\ x'$
using *std-apart-apart'*[*of x - ''1''*] **unfolding** $vars_{ls}$*-def* **by** *auto*
moreover
from *xin* **have** $x \in vars_{ls}\ (std_2\ C_2)$ **by** *auto*
then have $\exists x'.\ x{=}\ ''2''\ @x'$
using *std-apart-apart'*[*of x - ''2''*] **unfolding** $vars_{ls}$*-def* **by** *auto*
ultimately have *False* **by** *auto*
then have $x \in \{\}$ **by** *auto*
}
then show *?thesis* **by** *auto*
qed

lemma *std-apart-instance-of*$_{ls}$*1*: $instance\text{-}of_{ls}\ C_1\ (std_1\ C_1)$
proof −
have *empty*: $(\lambda x.\ Var\ (''1''@x)) \cdot (\lambda x.\ Var\ (tl\ x)) = \varepsilon$ **using** *composition-def*
by *auto*

have $C_1 \cdot_{ls} \varepsilon = C_1$ **using** *empty-subls* **by** *auto*
then have $C_1 \cdot_{ls} ((\lambda x.\ Var\ (''1''@x)) \cdot (\lambda x.\ Var\ (tl\ x))) = C_1$ **using** *empty* **by**
auto
then have $(C_1 \cdot_{ls} (\lambda x.\ Var\ (''1''@x))) \cdot_{ls} (\lambda x.\ Var\ (tl\ x)) = C_1$ **using** *compo-*
sition-conseq2ls **by** *auto*
then have $C_1 = (std_1\ C_1) \cdot_{ls} (\lambda x.\ Var\ (tl\ x))$ **by** *auto*
then show $instance\text{-}of_{ls}\ C_1\ (std_1\ C_1)$ **unfolding** $instance\text{-}of_{ls}$*-def* **by** *auto*
qed

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

have *empty*: $(\lambda x.\ Var\ (''2''@x)) \cdot (\lambda x.\ Var\ (tl\ x)) = \varepsilon$ **using** *composition-def* **by** *auto*

have $C2 \cdot_{ls} \varepsilon = C2$ **using** *empty-subls* **by** *auto*
then have $C2 \cdot_{ls} ((\lambda x.\ Var\ (''2''@x)) \cdot (\lambda x.\ Var\ (tl\ x))) = C2$ **using** *empty* **by** *auto*
then have $(C2 \cdot_{ls} (\lambda x.\ Var\ (''2''@x))) \cdot_{ls} (\lambda x.\ Var\ (tl\ x)) = C2$ **using** *composition-conseq2ls* **by** *auto*
then have $C2 = (std_2\ C2) \cdot_{ls} (\lambda x.\ Var\ (tl\ x))$ **by** *auto*
then show $instance\text{-}of_{ls}\ C2\ (std_2\ C2)$ **unfolding** $instance\text{-}of_{ls}\text{-}def$ **by** *auto*
qed

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 −
from *nonempty* **obtain** l **where** $l \in L$ **by** *auto*
from *unif this* **have** $L \cdot_{ls} \sigma = \{l \cdot_l \sigma\}$ **unfolding** $unifier_{ls}\text{-}def$ **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *unifiert-def2*:
assumes *L-elem*: $ts \neq \{\}$
shows $unifier_{ts}\ \sigma\ ts \longleftrightarrow (\exists l.\ (\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{l\})$
proof
assume *unif*: $unifier_{ts}\ \sigma\ ts$
from *L-elem* **obtain** t **where** $t \in ts$ **by** *auto*
then have $(\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{t \cdot_t \sigma\}$ **using** *unif* **unfolding** $unifier_{ts}\text{-}def$ **by** *auto*
then show $\exists l.\ (\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{l\}$ **by** *auto*
next
assume $\exists l.\ (\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{l\}$
then obtain l **where** $(\lambda t.\ sub\ t\ \sigma)\ `\ ts = \{l\}$ **by** *auto*
then have $\forall l' \in ts.\ l' \cdot_t \sigma = l$ **by** *auto*
then show $unifier_{ts}\ \sigma\ ts$ **unfolding** $unifier_{ts}\text{-}def$ **by** *auto*
qed

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

proof
assume *unif*: $unifier_{ls}\ \sigma\ L$
from *L-elem* **obtain** l **where** $l \in L$ **by** *auto*
then have $L \cdot_{ls} \sigma = \{l \cdot_l \sigma\}$ **using** *unif* **unfolding** $unifier_{ls}$*-def* **by** *auto*
then show $\exists\, l.\ L \cdot_{ls} \sigma = \{l\}$ **by** *auto*
next
assume $\exists\, l.\ L \cdot_{ls} \sigma = \{l\}$
then obtain l **where** $L \cdot_{ls} \sigma = \{l\}$ **by** *auto*
then have $\forall\, l' \in L.\ l' \cdot_l \sigma = l$ **by** *auto*
then show $unifier_{ls}\ \sigma\ L$ **unfolding** $unifier_{ls}$*-def* **by** *auto*
qed

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 −
from *unif empt* **have** $\exists\, l.\ L \cdot_{ls} \sigma' = \{l\}$ **using** *unif-sub* **by** *auto*
then show *?thesis* **using** $ground_{ls}$*-subls* $ground_{ls}$ **by** *auto*
qed

definition *unifiablets* :: *fterm set* ⇒ *bool* **where**
unifiablets fs ⟷ $(\exists \sigma.\ unifier_{ts}\ \sigma\ fs)$

definition *unifiablels* :: *fterm literal set* ⇒ *bool* **where**
unifiablels L ⟷ $(\exists \sigma.\ unifier_{ls}\ \sigma\ L)$

lemma *unifier-comp*[*simp*]: $unifier_{ls}\ \sigma\ (L^C) \longleftrightarrow unifier_{ls}\ \sigma\ L$
proof
assume $unifier_{ls}\ \sigma\ (L^C)$
then obtain l'' **where** l''*-p*: $\forall\, l \in L^C.\ l \cdot_l \sigma = l''$
unfolding $unifier_{ls}$*-def* **by** *auto*
obtain l' **where** $(l')^c = l''$ **using** *comp-exi2*[*of* l''] **by** *auto*
from *this* l''*-p* **have** l'*-p*:$\forall\, l \in L^C.\ l \cdot_l \sigma = (l')^c$ **by** *auto*
have $\forall\, l \in L.\ l \cdot_l \sigma = l'$
proof
fix l
assume $l \in L$
then have $l^c \in L^C$ **by** *auto*
then have $(l^c) \cdot_l \sigma = (l')^c$ **using** l'*-p* **by** *auto*
then have $(l \cdot_l \sigma)^c = (l')^c$ **by** (*cases l*) *auto*
then show $l \cdot_l \sigma = l'$ **using** *cancel-comp2* **by** *blast*
qed
then show $unifier_{ls}\ \sigma\ L$ **unfolding** $unifier_{ls}$*-def* **by** *auto*
next
assume $unifier_{ls}\ \sigma\ L$
then obtain l' **where** l'*-p*: $\forall\, l \in L.\ l \cdot_l \sigma = l'$ **unfolding** $unifier_{ls}$*-def* **by** *auto*
have $\forall\, l \in L^C.\ l \cdot_l \sigma = (l')^c$

proof
fix l
assume $l \in L^C$
then have $l^c \in L$ **using** *cancel-comp1* **by** (*metis image-iff*)
then show $l \cdot_l \sigma = (l')^c$ **using** l'*-p comp-sub cancel-comp1* **by** *metis*
qed
then show $unifier_{ls}\ \sigma\ (L^C)$ **unfolding** $unifier_{ls}$*-def* **by** *auto*
qed

lemma *unifier-sub1*:
assumes $unifier_{ls}\ \sigma\ L$
assumes $L' \subseteq L$
shows $unifier_{ls}\ \sigma\ L'$
using *assms* **unfolding** $unifier_{ls}$*-def* **by** *auto*

lemma *unifier-sub2*:
assumes *asm*: $unifier_{ls}\ \sigma\ (L_1 \cup L_2)$
shows $unifier_{ls}\ \sigma\ L_1 \wedge unifier_{ls}\ \sigma\ L_2$
proof −
have $L_1 \subseteq (L_1 \cup L_2) \wedge L_2 \subseteq (L_1 \cup L_2)$ **by** *simp*
from *this asm* **show** *?thesis* **using** *unifier-sub1* **by** *auto*
qed

9.1 Most General Unifiers

definition mgu_{ts} :: *substitution* ⇒ *fterm set* ⇒ *bool* **where**
$mgu_{ts}\ \sigma\ ts \longleftrightarrow unifier_{ts}\ \sigma\ ts \wedge (\forall\, u.\ unifier_{ts}\ u\ ts \longrightarrow (\exists\, i.\ u = \sigma \cdot i))$

definition mgu_{ls} :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
$mgu_{ls}\ \sigma\ L \longleftrightarrow unifier_{ls}\ \sigma\ L \wedge (\forall\, u.\ unifier_{ls}\ u\ L \longrightarrow (\exists\, i.\ u = \sigma \cdot i))$

10 Resolution

definition *applicable* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *bool* **where**
applicable $C_1\ C_2\ L_1\ L_2\ \sigma \longleftrightarrow$
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
$\wedge\ vars_{ls}\ C_1 \cap vars_{ls}\ C_2 = \{\}$
$\wedge\ L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
$\wedge\ mgu_{ls}\ \sigma\ (L_1 \cup {L_2}^C)$

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

definition *resolution* :: *fterm clause* ⇒ *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:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *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* $\Rightarrow$ *fterm clause set* $\Rightarrow$ *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* $\Rightarrow$ *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 $E\ F\ \sigma$ = $eval_t\ E\ F \circ \sigma$

lemma *substitutiont*: $eval_t\ E\ F\ (t \cdot_t \sigma) = eval_t$ (*evalsub* $E\ F\ \sigma$) $F\ t$
apply (*induction t*)
unfolding *evalsub-def* **apply** *auto*
apply (*metis* (*mono-tags*, *lifting*) *comp-apply map-cong*)
done

lemma *substitutionts*: $eval_{ts}\ E\ F\ (ts \cdot_{ts} \sigma) = eval_{ts}$ (*evalsub* $E\ F\ \sigma$) $F\ ts$
using *substitutiont* **by** *auto*

lemma *substitution*: $eval_l\ E\ F\ G\ (l \cdot_l \sigma) \longleftrightarrow eval_l$ (*evalsub* $E\ F\ \sigma$) $F\ G\ l$
apply (*induction l*)
using *substitutionts* **apply** (*metis* $eval_l$*.simps*(*1*) *subl.simps*(*1*))
using *substitutionts* **apply** (*metis* $eval_l$*.simps*(*2*) *subl.simps*(*2*))
done

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

```
  fix E
  from asm have ∀ E'. ∃ l ∈ C. eval_l E' F G l using eval_c-def by blast
  then have ∃ l ∈ C. eval_l (evalsub E F σ) F G l by auto
  then show ∃ l ∈ C ·_ls σ. eval_l E F G l using substitution by blast
qed

lemma simple-resolution-sound:
  assumes C_1sat: eval_c F G C_1
  assumes C_2sat: eval_c F G C_2
  assumes l_1inc_1: l_1 ∈ C_1
  assumes l_2inc_2: l_2 ∈ C_2
  assumes comp: l_1^c = l_2
  shows eval_c F G ((C_1 − {l_1}) ∪ (C_2 − {l_2}))
proof −
  have ∀ E. ∃ l ∈ (((C_1 − {l_1}) ∪ (C_2 − {l_2}))). eval_l E F G l
    proof
      fix E
      have eval_l E F G l_1 ∨ eval_l E F G l_2 using comp by (cases l_1) auto
      then show ∃ l ∈ (((C_1 − {l_1}) ∪ (C_2 − {l_2}))). eval_l E F G l
        proof
          assume eval_l E F G l_1
          then have ¬eval_l E F G l_2 using comp by (cases l_1) auto
            then have ∃ l_2'∈ C_2. l_2' ≠ l_2 ∧ eval_l E F G l_2' using l_2inc_2 C_2sat
unfolding eval_c-def by auto
          then show ∃ l∈(C_1 − {l_1}) ∪ (C_2 − {l_2}). eval_l E F G l by auto
        next
          assume eval_l E F G l_2
          then have ¬eval_l E F G l_1 using comp by (cases l_1) auto
            then have ∃ l_1'∈ C_1. l_1' ≠ l_1 ∧ eval_l E F G l_1' using l_1inc_1 C_1sat
unfolding eval_c-def by auto
          then show ∃ l∈(C_1 − {l_1}) ∪ (C_2 − {l_2}). eval_l E F G l by auto
        qed
    qed
  then show ?thesis unfolding eval_c-def by simp
qed

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 σ
  shows eval_c F G (mresolution C_1 C_2 L_1 L_2 σ)
proof −
  from sat_1 have sat_1σ: eval_c F G (C_1 ·_ls σ) using subst-sound by blast
  from sat_2 have sat_2σ: eval_c F G (C_2 ·_ls σ) using subst-sound by blast

  from appl obtain l_1 where l_1-p: l_1 ∈ L_1 unfolding applicable-def by auto

  from l_1-p appl have l_1 ∈ C_1 unfolding applicable-def by auto
  then have inc_1σ: l_1 ·_l σ ∈ C_1 ·_ls σ by auto
```

from l_1-*p* **have** *unified*$_1$: $l_1 \in (L_1 \cup ({L_2}^C))$ **by** *auto*

from l_1-*p appl* **have** $l_1\sigma isl_1\sigma$: $\{l_1 \cdot_l \sigma\} = L_1 \cdot_{ls} \sigma$
unfolding *mgu*$_{ls}$-*def unifier*$_{ls}$-*def applicable-def* **by** *auto*

from *appl* **obtain** l_2 **where** l_2-*p*: $l_2 \in L_2$ **unfolding** *applicable-def* **by** *auto*

from l_2-*p appl* **have** $l_2 \in C_2$ **unfolding** *applicable-def* **by** *auto*
then have *inc*$_2\sigma$: $l_2 \cdot_l \sigma \in C_2 \cdot_{ls} \sigma$ **by** *auto*

from l_2-*p* **have** *unified*$_2$: ${l_2}^c \in (L_1 \cup ({L_2}^C))$ **by** *auto*

from *unified*$_1$ *unified*$_2$ *appl* **have** $l_1 \cdot_l \sigma = ({l_2}^c) \cdot_l \sigma$
unfolding *mgu*$_{ls}$-*def unifier*$_{ls}$-*def applicable-def* **by** *auto*
then have *comp*: $(l_1 \cdot_l \sigma)^c = l_2 \cdot_l \sigma$ **using** *comp-sub comp-swap* **by** *auto*

from *appl* **have** *unifier*$_{ls}$ σ $({L_2}^C)$
using *unifier-sub2* **unfolding** *mgu*$_{ls}$-*def applicable-def* **by** *blast*
then have *unifier*$_{ls}$ σ L_2 **by** *auto*
from *this* l_2-*p* **have** $l_2\sigma isl_2\sigma$: $\{l_2 \cdot_l \sigma\} = L_2 \cdot_{ls} \sigma$ **unfolding** *unifier*$_{ls}$-*def* **by** *auto*

from *sat*$_1\sigma$ *sat*$_2\sigma$ *inc*$_1\sigma$ *inc*$_2\sigma$ *comp* **have** *eval*$_c$ *F G* $((C_1 \cdot_{ls} \sigma) - \{l_1 \cdot_l \sigma\} \cup ((C_2 \cdot_{ls} \sigma) - \{l_2 \cdot_l \sigma\}))$ **using** *simple-resolution-sound*[*of F G* $C_1 \cdot_{ls} \sigma$ $C_2 \cdot_{ls} \sigma$ $l_1 \cdot_l \sigma$ $l_2 \cdot_l \sigma$]
by *auto*
from *this* $l_1\sigma isl_1\sigma$ $l_2\sigma isl_2\sigma$ **show** *?thesis* **unfolding** *mresolution-def* **by** *auto*
qed

lemma *resolution-superset*: *mresolution* C_1 C_2 L_1 L_2 $\sigma \subseteq$ *resolution* C_1 C_2 L_1 L_2 σ
unfolding *mresolution-def resolution-def* **by** *auto*

lemma *superset-sound*:
assumes *sup*: $C \subseteq C'$
assumes *sat*: *eval*$_c$ *F G C*
shows *eval*$_c$ *F G* C'
proof −
have $\forall E.\ \exists l \in C'.$ *eval*$_l$ *E F G l*
proof
fix *E*
from *sat* **have** $\forall E.\ \exists l \in C.$ *eval*$_l$ *E F G l* **unfolding** *eval*$_c$-*def* **by** −
then have $\exists l \in C$. *eval*$_l$ *E F G l* **by** *auto*
then show $\exists l \in C'.$ *eval*$_l$ *E F G l* **using** *sup* **by** *auto*
qed
then show *eval*$_c$ *F G* C' **unfolding** *eval*$_c$-*def* **by** *auto*
qed

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 −
from $sat_1\ sat_2$ *appl* **have** $eval_c\ F\ G$ (*mresolution* $C_1\ C_2\ L_1\ L_2\ \sigma$) **using** *mresolution-sound* **by** *blast*
then show *?thesis* **using** *superset-sound resolution-superset* **by** *metis*
qed

lemma *mstep-sound*:
assumes *mresolution-step Cs Cs′*
assumes $eval_{cs}\ F\ G\ Cs$
shows $eval_{cs}\ F\ G\ Cs'$
using *assms* **proof** (*induction rule*: *mresolution-step.induct*)
case (*mresolution-rule* $C_1\ Cs\ C_2\ l_1\ l_2\ \sigma$)
then have $eval_c\ F\ G\ C_1 \wedge eval_c\ F\ G\ C_2$ **unfolding** $eval_{cs}$*-def* **by** *auto*
then have $eval_c\ F\ G$ (*mresolution* $C_1\ C_2\ l_1\ l_2\ \sigma$)
using *mresolution-sound mresolution-rule* **by** *auto*
then show *?case* **using** *mresolution-rule* **unfolding** $eval_{cs}$*-def* **by** *auto*
next
case (*standardize-apart C Cs C′*)
then have $eval_c\ F\ G\ C$ **unfolding** $eval_{cs}$*-def* **by** *auto*
then have $eval_c\ F\ G\ C'$ **using** *subst-sound standardize-apart* **unfolding** *var-renaming-of-def* $instance\text{-}of_{ls}$*-def* **by** *metis*
then show *?case* **using** *standardize-apart* **unfolding** $eval_{cs}$*-def* **by** *auto*
qed

theorem *step-sound*:
assumes *resolution-step Cs Cs′*
assumes $eval_{cs}\ F\ G\ Cs$
shows $eval_{cs}\ F\ G\ Cs'$
using *assms* **proof** (*induction rule*: *resolution-step.induct*)
case (*resolution-rule* $C_1\ Cs\ C_2\ l_1\ l_2\ \sigma$)
then have $eval_c\ F\ G\ C_1 \wedge eval_c\ F\ G\ C_2$ **unfolding** $eval_{cs}$*-def* **by** *auto*
then have $eval_c\ F\ G$ (*resolution* $C_1\ C_2\ l_1\ l_2\ \sigma$)
using *resolution-sound resolution-rule* **by** *auto*
then show *?case* **using** *resolution-rule* **unfolding** $eval_{cs}$*-def* **by** *auto*
next
case (*standardize-apart C Cs C′*)
then have $eval_c\ F\ G\ C$ **unfolding** $eval_{cs}$*-def* **by** *auto*
then have $eval_c\ F\ G\ C'$ **using** *subst-sound standardize-apart* **unfolding** *var-renaming-of-def* $instance\text{-}of_{ls}$*-def* **by** *metis*
then show *?case* **using** *standardize-apart* **unfolding** $eval_{cs}$*-def* **by** *auto*
qed

lemma *mderivation-sound*:
assumes *mresolution-deriv Cs Cs′*

assumes $eval_{cs}\ F\ G\ Cs$
shows $eval_{cs}\ F\ G\ Cs'$
using *assms* **unfolding** *mresolution-deriv-def*
proof (*induction rule*: *rtranclp.induct*)
case *rtrancl-refl* **then show** *?case* **by** *auto*
next
case (*rtrancl-into-rtrancl* $Cs_1\ Cs_2\ Cs_3$) **then show** *?case* **using** *mstep-sound* **by** *auto*
qed

theorem *derivation-sound*:
assumes *resolution-deriv Cs Cs'*
assumes $eval_{cs}\ F\ G\ Cs$
shows$eval_{cs}\ F\ G\ Cs'$
using *assms* **unfolding** *resolution-deriv-def*
proof (*induction rule*: *rtranclp.induct*)
case *rtrancl-refl* **then show** *?case* **by** *auto*
next
case (*rtrancl-into-rtrancl* $Cs_1\ Cs_2\ Cs_3$) **then show** *?case* **using** *step-sound* **by** *auto*
qed

theorem *derivation-sound-refute*:
assumes *deriv*: *resolution-deriv Cs Cs'* $\wedge$ $\{\} \in Cs'$
shows $\neg eval_{cs}\ F\ G\ Cs$
proof −
from *deriv* **have** $eval_{cs}\ F\ G\ Cs \longrightarrow eval_{cs}\ F\ G\ Cs'$ **using** *derivation-sound* **by** *auto*
moreover
from *deriv* **have** $eval_{cs}\ F\ G\ Cs' \longrightarrow eval_c\ F\ G\ \{\}$ **unfolding** $eval_{cs}$*-def* **by** *auto*
moreover
then have $eval_c\ F\ G\ \{\} \longrightarrow False$ **unfolding** $eval_c$*-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

12 Herbrand Interpretations

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

term *HFun*

lemma $eval\text{-}ground_t$:
assumes $ground_t\ t$
shows $(eval_t\ E\ HFun\ t) = hterm\text{-}of\text{-}fterm\ t$
using *assms* **by** (*induction t*) *auto*

lemma $eval\text{-}ground_{ts}$:

assumes $ground_{ts}\ ts$
shows $(eval_{ts}\ E\ HFun\ ts) = hterms\text{-}of\text{-}fterms\ ts$
unfolding *hterms-of-fterms-def* **using** *assms* $eval\text{-}ground_t$ **by** (*induction ts*) *auto*

lemma $eval_l\text{-}ground_{ts}$:
assumes *asm*: $ground_{ts}\ ts$
shows $eval_l\ E\ HFun\ G\ (Pos\ P\ ts) \longleftrightarrow G\ P\ (hterms\text{-}of\text{-}fterms\ ts)$
proof −
have $eval_l\ E\ HFun\ G\ (Pos\ P\ ts) = G\ P\ (eval_{ts}\ E\ HFun\ ts)$ **by** *auto*
also have ... $= G\ P\ (hterms\text{-}of\text{-}fterms\ ts)$ **using** *asm* $eval\text{-}ground_{ts}$ **by** *simp*
finally show *?thesis* **by** *auto*
qed

13 Partial Interpretations

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

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

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

abbreviation $falsifies_g$ :: *partial-pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
$falsifies_g\ G\ C \equiv ground_{ls}\ C \wedge (\forall\, l \in C.\ falsifies_l\ G\ l)$

abbreviation $falsifies_c$ :: *partial-pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
$falsifies_c\ G\ C \equiv (\exists\, C'.\ instance\text{-}of_{ls}\ C'\ C \wedge falsifies_g\ G\ C')$

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*: $ground_t\ (t \cdot_t (sub\text{-}of\text{-}denot\ E))$
by (*induction t*) (*auto simp add*: *ground-fterm-of-hterm*)

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

using *ground-sub-of-denott* **by** *simp*

lemma *ground-sub-of-denotl*: $ground_l\ (l \cdot_l sub\text{-}of\text{-}denot\ E)$
proof −
  have $ground_{ts}\ (subs\ (get\text{-}terms\ l)\ (sub\text{-}of\text{-}denot\ E))$
    using *ground-sub-of-denotts* **by** *auto*
  then show *?thesis* **by** (*cases l*) *auto*
qed

lemma *sub-of-denot-equivx*: $eval_t\ E\ HFun\ (sub\text{-}of\text{-}denot\ E\ x) = E\ x$
proof −
  have $ground_t\ (sub\text{-}of\text{-}denot\ E\ x)$ **using** *ground-fterm-of-hterm* **by** *simp*
  then
  have $eval_t\ E\ HFun\ (sub\text{-}of\text{-}denot\ E\ x) = hterm\text{-}of\text{-}fterm\ (sub\text{-}of\text{-}denot\ E\ x)$
    using $eval\text{-}ground_t(1)$ **by** *auto*
  also have ... = $hterm\text{-}of\text{-}fterm\ (fterm\text{-}of\text{-}hterm\ (E\ x))$ **by** *auto*
  also have ... = $E\ x$ **by** *auto*
  finally show *?thesis* **by** *auto*
qed

lemma *sub-of-denot-equivt*:
    $eval_t\ E\ HFun\ (t \cdot_t (sub\text{-}of\text{-}denot\ E)) = eval_t\ E\ HFun\ t$
using *sub-of-denot-equivx* **by** (*induction t*) *auto*

lemma *sub-of-denot-equivts*: $eval_{ts}\ E\ HFun\ (ts \cdot_{ts} (sub\text{-}of\text{-}denot\ E)) = eval_{ts}\ E\ HFun\ ts$
using *sub-of-denot-equivt* **by** *simp*

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 (*induction l*)
  case (*Pos p ts*)
  have $eval_l\ E\ HFun\ G\ ((Pos\ p\ ts) \cdot_l sub\text{-}of\text{-}denot\ E) \longleftrightarrow G\ p\ (eval_{ts}\ E\ HFun\ (ts \cdot_{ts} (sub\text{-}of\text{-}denot\ E)))$ **by** *auto*
  also have ... $\longleftrightarrow G\ p\ (eval_{ts}\ E\ HFun\ ts)$ **using** *sub-of-denot-equivts*[*of E ts*] **by** *metis*
  also have ... $\longleftrightarrow eval_l\ E\ HFun\ G\ (Pos\ p\ ts)$ **by** *simp*
  finally
  show *?case* **by** *blast*
next
 case (*Neg p ts*)
  have $eval_l\ E\ HFun\ G\ ((Neg\ p\ ts) \cdot_l sub\text{-}of\text{-}denot\ E) \longleftrightarrow \neg G\ p\ (eval_{ts}\ E\ HFun\ (ts \cdot_{ts} (sub\text{-}of\text{-}denot\ E)))$ **by** *auto*
  also have ... $\longleftrightarrow \neg G\ p\ (eval_{ts}\ E\ HFun\ ts)$ **using** *sub-of-denot-equivts*[*of E ts*] **by** *metis*
  also have ... = $eval_l\ E\ HFun\ G\ (Neg\ p\ ts)$ **by** *simp*
  finally
  show *?case* **by** *blast*

qed

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* ·$_l$ *sub-of-denot E*) ∧ *ground*$_l$ (*l* ·$_l$ *sub-of-denot E*)
using *sub-of-denot-equivl ground-sub-of-denotl* **by** *auto*

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

lemma *partial-equiv-subst*:
assumes *falsifies*$_c$ *G* (*C* ·$_{ls}$ *τ*)
shows *falsifies*$_c$ *G C*
proof −
from *assms* **obtain** *C′* **where** *C′-p*: *instance-of*$_{ls}$ *C′* (*C* ·$_{ls}$ *τ*) ∧ *falsifies*$_g$ *G C′* **by** *auto*
then have *instance-of*$_{ls}$ (*C* ·$_{ls}$ *τ*) *C* **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
then have *instance-of*$_{ls}$ *C′ C* **using** *C′-p instance-of*$_{ls}$*-trans* **by** *auto*
then show *?thesis* **using** *C′-p* **by** *auto*
qed

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

lemma *sub-of-denot-equiv-ground*:
((∃ *l* ∈ *C*. *eval*$_l$ *E HFun G l*) ⟷ (∃ *l* ∈ *C* ·$_{ls}$ *sub-of-denot E*. *eval*$_l$ *E HFun G l*))
∧ *ground*$_{ls}$ (*C* ·$_{ls}$ *sub-of-denot E*)
using *sub-of-denot-equiv-ground′* **by** *auto*

lemma *std*$_1$*-falsifies*: *falsifies*$_c$ *G C*$_1$ ⟷ *falsifies*$_c$ *G* (*std*$_1$ *C*$_1$)
proof
assume *asm*: *falsifies*$_c$ *G C*$_1$
then obtain *Cg* **where** *instance-of*$_{ls}$ *Cg C*$_1$ ∧ *falsifies*$_g$ *G Cg* **by** *auto*
moreover
then have *instance-of*$_{ls}$ *Cg* (*std*$_1$ *C*$_1$) **using** *std-apart-instance-of*$_{ls}$*1 instance-of*$_{ls}$*-trans* **by** *blast*
ultimately
show *falsifies*$_c$ *G* (*std*$_1$ *C*$_1$) **by** *auto*
next
assume *asm*: *falsifies*$_c$ *G* (*std*$_1$ *C*$_1$)
then have *inst*: *instance-of*$_{ls}$ (*std*$_1$ *C*$_1$) *C*$_1$ **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*

from *asm* **obtain** *Cg* **where** *instance-of*$_{ls}$ *Cg* (*std*$_1$ *C*$_1$) ∧ *falsifies*$_g$ *G Cg* **by** *auto*
moreover
then have *instance-of*$_{ls}$ *Cg C*$_1$ **using** *inst instance-of*$_{ls}$*-trans* **by** *blast*
ultimately

show *falsifies*$_c$ *G* C_1 **by** *auto*
qed

lemma *std*$_2$*-falsifies*: *falsifies*$_c$ *G* $C_2 \longleftrightarrow$ *falsifies*$_c$ *G* (*std*$_2$ C_2)
proof
assume *asm*: *falsifies*$_c$ *G* C_2
then obtain *Cg* **where** *instance-of*$_{ls}$ *Cg* C_2 $\wedge$ *falsifies*$_g$ *G Cg* **by** *auto*
moreover
then have *instance-of*$_{ls}$ *Cg* (*std*$_2$ C_2) **using** *std-apart-instance-of*$_{ls}$*2 instance-of*$_{ls}$*-trans*
by *blast*
ultimately
show *falsifies*$_c$ *G* (*std*$_2$ C_2) **by** *auto*
next
assume *asm*: *falsifies*$_c$ *G* (*std*$_2$ C_2)
then have *inst*: *instance-of*$_{ls}$ (*std*$_2$ C_2) C_2 **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*

from *asm* **obtain** *Cg* **where** *instance-of*$_{ls}$ *Cg* (*std*$_2$ C_2) $\wedge$ *falsifies*$_g$ *G Cg* **by**
auto
moreover
then have *instance-of*$_{ls}$ *Cg* C_2 **using** *inst instance-of*$_{ls}$*-trans* **by** *blast*
ultimately
show *falsifies*$_c$ *G* C_2 **by** *auto*
qed

lemma *std*$_1$*-renames*: *var-renaming-of* C_1 (*std*$_1$ C_1)
proof −
have *instance-of*$_{ls}$ C_1 (*std*$_1$ C_1) **using** *std-apart-instance-of*$_{ls}$*1* **by** *auto*
moreover have *instance-of*$_{ls}$ (*std*$_1$ C_1) C_1 **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
ultimately show *var-renaming-of* C_1 (*std*$_1$ C_1) **unfolding** *var-renaming-of-def*
by *auto*
qed

lemma *std*$_2$*-renames*: *var-renaming-of* C_2 (*std*$_2$ C_2)
proof −
have *instance-of*$_{ls}$ C_2 (*std*$_2$ C_2) **using** *std-apart-instance-of*$_{ls}$*2* **by** *auto*
moreover have *instance-of*$_{ls}$ (*std*$_2$ C_2) C_2 **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
ultimately show *var-renaming-of* C_2 (*std*$_2$ C_2) **unfolding** *var-renaming-of-def*
by *auto*
qed

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
from *asm* **show** ∀ *l*∈*C*. *f l* ≤ (*Max* (*f* ' *C*)) **by** *auto*
qed

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*: ∀ *l*∈*C*. *nat-of-fatom* (*get-atom l*) ≤ *n*
shows *eval*$_{c}$ *HFun* (*extend f*) *C*
proof −
let *?F* = *HFun*
let *?G* = *extend f*
{
fix *E*
from *C-sat* **have** ∀ *C'*. (¬*instance-of*$_{ls}$ *C' C* ∨ ¬*falsifies*$_{g}$ (*f* (*Suc n*)) *C'*) **by** *auto*
then have ¬*falsifies*$_{g}$ (*f* (*Suc n*)) *C* **using** *instance-of*$_{ls}$*-self* **by** *auto*
then obtain *l* **where** *l-p*: *l*∈*C* ∧ ¬*falsifies*$_{l}$ (*f* (*Suc n*)) *l* **using** *C-ground* **by** *blast*
let *?i* = *nat-of-fatom* (*get-atom l*)

from *l-p* **have** *i-n*: *?i* ≤ *n* **using** *n-max* **by** *auto*
then have *j-n*: *?i* < *length* (*f* (*Suc n*)) **using** *f-infpath infpath-length*[*of f*] **by** *auto*

have *eval*$_{l}$ *E HFun* (*extend f*) *l*
proof (*cases l*)
case (*Pos P ts*)
from *Pos l-p C-ground* **have** *ts-ground*: *ground*$_{ts}$ *ts* **by** *auto*

have ¬*falsifies*$_{l}$ (*f* (*Suc n*)) *l* **using** *l-p* **by** *auto*
then have *f* (*Suc n*) ! *?i* = *True*
using *j-n Pos ts-ground empty-subts*[*of ts*] **unfolding** *falsifies*$_{l}$*-def* **by** *auto*
moreover have *f* (*Suc ?i*) ! *?i* = *f* (*Suc n*) ! *?i*
using *f-infpath i-n j-n infpath-length*[*of f*] *ith-in-extension*[*of f*] **by** *simp*
ultimately
have *f* (*Suc ?i*) ! *?i* = *True* **using** *Pos* **by** *auto*
then have *?G P* (*hterms-of-fterms ts*) **using** *Pos* **by** (*simp add*: *nat-of-fatom-def*)

```
      then show ?thesis using eval_l-ground_ts[of ts - ?G P] ts-ground Pos by
auto
    next
      case (Neg P ts)
      from Neg l-p C-ground have ts-ground: ground_ts ts by auto

      have ¬falsifies_l (f (Suc n)) l using l-p by auto
      then have f (Suc n) ! ?i = False
       using j-n Neg ts-ground empty-subts[of ts] unfolding falsifies_l-def by auto
      moreover have f (Suc ?i) ! ?i = f (Suc n) ! ?i
        using f-infpath i-n j-n infpath-length[of f] ith-in-extension[of f] by simp
      ultimately
      have f (Suc ?i) ! ?i = False using Neg by auto
    then have ¬?G P (hterms-of-fterms ts) using Neg by (simp add: nat-of-fatom-def)

      then show ?thesis using Neg eval_l-ground_ts[of ts - ?G P] ts-ground by
auto
    qed
  then have ∃ l ∈ C. eval_l E HFun (extend f) l using l-p by auto
 }
 then have eval_c HFun (extend f) C unfolding eval_c-def by auto
 then show ?thesis using instance-of_ls-self by auto
qed

lemma extend-preserves-model2:
 assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
 assumes C-ground: ground_ls C
 assumes fin-c: finite C
 assumes model-C: ∀ n. ¬falsifies_c (f n) C
 shows C-false: eval_c HFun (extend f) C
proof −
 — Since C is finite, C has a largest index of a literal.
  obtain n where largest: ∀ l ∈ C. nat-of-fatom (get-atom l) ≤ n using fin-c
maximum[of C λl. nat-of-fatom (get-atom l)] by blast
 moreover
 then have ¬falsifies_c (f (Suc n)) C using model-C by auto
 ultimately show ?thesis using model-C f-infpath C-ground extend-preserves-model[of
f C n ] by blast
qed

lemma extend-infpath:
 assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
 assumes model-c: ∀ n. ¬falsifies_c (f n) C
 assumes fin-c: finite C
 shows eval_c HFun (extend f) C
unfolding eval_c-def proof
 fix E
 let ?G = extend f
```

let *?σ* = *sub-of-denot E*

from *fin-c* **have** *fin-cσ*: *finite* (*C* ·$_{ls}$ *sub-of-denot E*) **by** *auto*
have *groundcσ*: *ground*$_{ls}$ (*C* ·$_{ls}$ *sub-of-denot E*) **using** *sub-of-denot-equiv-ground* **by** *auto*

— Here starts the proof
— We go from syntactic FO world to syntactic ground world:
from *model-c* **have** ∀ *n*. ¬*falsifies*$_c$ (*f n*) (*C* ·$_{ls}$ *?σ*) **using** *partial-equiv-subst* **by** *blast*
— Then from syntactic ground world to semantic ground world:
then have *eval*$_c$ *HFun ?G* (*C* ·$_{ls}$ *?σ*) **using** *groundcσ f-infpath fin-cσ extend-preserves-model2*[*of f C* ·$_{ls}$ *?σ*] **by** *blast*
— Then from semantic ground world to semantic FO world:
then have ∀ *E*. ∃ *l* ∈ (*C* ·$_{ls}$ *?σ*). *eval*$_l$ *E HFun ?G l* **unfolding** *eval*$_c$*-def* **by** *auto*
then have ∃ *l* ∈ (*C* ·$_{ls}$ *?σ*). *eval*$_l$ *E HFun ?G l* **by** *auto*
then show ∃ *l* ∈ *C*. *eval*$_l$ *E HFun ?G l* **using** *sub-of-denot-equiv-ground*[*of C E extend f*] **by** *blast*
qed

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

lemma *infpath-model*:
assumes *f-infpath*: *wf-infpath* (*f* :: *nat* ⇒ *partial-pred-denot*)
assumes *model-cs*: ∀ *n*. ¬*falsifies*$_{cs}$ (*f n*) *Cs*
assumes *fin-cs*: *finite Cs*
assumes *fin-c*: ∀ *C* ∈ *Cs*. *finite C*
shows *eval*$_{cs}$ *HFun* (*extend f*) *Cs*
proof −
let *?F* = *HFun*

have ∀ *C* ∈ *Cs*. *eval*$_c$ *?F* (*extend f*) *C*
proof (*rule ballI*)
fix *C*
assume *asm*: *C* ∈ *Cs*
then have ∀ *n*. ¬*falsifies*$_c$ (*f n*) *C* **using** *model-cs* **by** *auto*
then show *eval*$_c$ *?F* (*extend f*) *C* **using** *fin-c asm f-infpath extend-infpath*[*of f C*] **by** *auto*
qed
then show *eval*$_{cs}$ *?F* (*extend f*) *Cs* **unfolding** *eval*$_{cs}$*-def* **by** *auto*
qed

fun *deeptree* :: *nat* ⇒ *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*

using *assms* **proof** (*induction n arbitrary*: *b*)
case *0* **then show** *?case* **using** *branch-inv-Leaf* **by** *auto*
next
case (*Suc n*)
then have *branch b* (*Branching* (*deeptree n*) (*deeptree n*)) **by** *auto*
then obtain *a b'* **where** *p*: *b* = *a*#*b'*∧ *branch b'* (*deeptree n*) **using** *branch-inv-Branching*[*of b*] **by** *blast*
then have *length b'* = *n* **using** *Suc* **by** *auto*
then show *?case* **using** *p* **by** *auto*
qed

lemma *infinity*:
assumes *inj*: ∀ *n* :: *nat*. *undiago* (*diago n*) = *n*
assumes *all-tree*: ∀ *n* :: *nat*. (*diago n*) ∈ *tree*
shows ¬*finite tree*
proof −
from *inj all-tree* **have** ∀ *n*. *n* = *undiago* (*diago n*) ∧ (*diago n*) ∈ *tree* **by** *auto*
then have ∀ *n*. ∃ *ds*. *n* = *undiago ds* ∧ *ds* ∈ *tree* **by** *auto*
then have *undiago* ' *tree* = (*UNIV* :: *nat set*) **by** *auto*
then have ¬*finite tree***by** (*metis finite-imageI infinite-UNIV-nat*)
then show *?thesis* **by** *auto*
qed

lemma *longer-falsifies*$_l$:
assumes *falsifies*$_l$ *ds l*
shows *falsifies*$_l$ (*ds*@*d*) *l*
proof −
let *?i* = *nat-of-fatom* (*get-atom l*)
from *assms* **have** *i-p*: *ground*$_l$ *l* ∧ *?i* < *length ds* ∧ *ds* ! *?i* = (¬*sign l*) **unfolding** *falsifies*$_l$*-def* **by** *meson*
moreover
from *i-p* **have** *?i* < *length* (*ds*@*d*) **by** *auto*
moreover
from *i-p* **have** (*ds*@*d*) ! *?i* = (¬*sign l*) **by** (*simp add*: *nth-append*)
ultimately
show *?thesis* **unfolding** *falsifies*$_l$*-def* **by** *simp*
qed

lemma *longer-falsifies*$_g$:
assumes *falsifies*$_g$ *ds C*
shows *falsifies*$_g$ (*ds* @ *d*) *C*
proof −
{
fix *l*
assume *l*∈*C*
then have *falsifies*$_l$ (*ds* @ *d*) *l* **using** *assms longer-falsifies*$_l$ **by** *auto*
} **then show** *?thesis* **using** *assms* **by** *auto*
qed

lemma *longer-falsifies$_c$*:
assumes *falsifies$_c$ ds C*
shows *falsifies$_c$ (ds @ d) C*
proof −
from *assms* **obtain** *C′* **where** *instance-of$_{ls}$ C′ C ∧ falsifies$_g$ ds C′* **by** *auto*
moreover
then have *falsifies$_g$ (ds @ d) C′* **using** *longer-falsifies$_g$* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

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 −
from *assms* **obtain** *C* **where** *C ∈ Cs ∧ falsifies$_c$ ds C* **by** *auto*
moreover
then have *falsifies$_c$ (ds @ d) C* **using** *longer-falsifies$_c$[of C ds d]* **by** *blast*
ultimately
show *?thesis* **by** *auto*
qed

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

theorem *herbrand′*:
assumes *openb*: *∀ T. ∃ G. open-branch G T Cs*
assumes *finite-cs*: *finite Cs ∀ C∈Cs. finite C*
shows *∃ G. eval$_{cs}$ HFun G Cs*
proof −
— Show T infinite:
let *?tree* = *{G. ¬falsifies$_{cs}$ G Cs}*
let *?undiag* = *length*
let *?diag* = *(λl. SOME b. open-branch b (deeptree l) Cs) :: nat ⇒ partial-pred-denot*

from *openb* **have** *diag-open*: *∀ l. open-branch (?diag l) (deeptree l) Cs*
using *someI-ex[of λb. open-branch b (deeptree -) Cs]* **by** *auto*
then have *∀ n. ?undiag (?diag n) = n* **using** *branch-length* **by** *auto*
moreover
have *∀ n. (?diag n) ∈ ?tree* **using** *diag-open* **by** *auto*
ultimately
have *¬finite ?tree* **using** *infinity[of - λn. SOME b. open-branch b (- n) Cs]* **by** *simp*
— Get infinite path:
moreover
have *∀ ds d. ¬falsifies$_{cs}$ (ds @ d) Cs ⟶ ¬falsifies$_{cs}$ ds Cs*
using *longer-falsifies[of Cs]* **by** *blast*
then have *(∀ ds d. ds @ d ∈ ?tree ⟶ ds ∈ ?tree)* **by** *auto*
ultimately
have *∃ c. wf-infpath c ∧ (∀ n. c n ∈ ?tree)* **using** *konig[of ?tree]* **by** *blast*

then have $\exists\, G.$ *wf-infpath* $G \wedge (\forall\, n.\ \neg$ *falsifies*$_{cs}$ $(G\ n)$ *Cs*) **by** *auto*
— Apply above infpath lemma:
then show $\exists\, G.$ *eval*$_{cs}$ *HFun G Cs* **using** *infpath-model finite-cs* **by** *auto*
qed

lemma *shorter-falsifies*$_l$:
assumes *falsifies*$_l$ $(ds@d)$ *l*
assumes *nat-of-fatom* $(get\text{-}atom\ l) <$ *length ds*
shows *falsifies*$_l$ *ds l*
proof −
let *?i* = *nat-of-fatom* (*get-atom l*)
from *assms* **have** *i-p*: *ground*$_l$ *l* ∧ *?i* < *length* $(ds@d) \wedge (ds@d)$! *?i* = $(\neg sign\ l)$ **unfolding** *falsifies*$_l$*-def* **by** *meson*
moreover
then have *?i* < *length ds* **using** *assms* **by** *auto*
moreover
then have *ds* ! *?i* = $(\neg sign\ l)$ **using** *i-p nth-append*[*of ds d ?i*] **by** *auto*
ultimately show *?thesis* **using** *assms* **unfolding** *falsifies*$_l$*-def* **by** *simp*
qed

theorem *herbrand′-contra*:
assumes *finite-cs*: *finite Cs* ∀ *C*∈*Cs*. *finite C*
assumes *unsat*: ∀ *G*. ¬*eval*$_{cs}$ *HFun G Cs*
shows $\exists\, T.\ \forall\, G.$ *branch G T* ⟶ *closed-branch G T Cs*
proof −
from *finite-cs unsat* **have** $(\forall\, T.\ \exists\, G.$ *open-branch G T Cs*$) \longrightarrow (\exists\, G.$ *eval*$_{cs}$ *HFun G Cs*) **using** *herbrand′* **by** *blast*
then show *?thesis* **using** *unsat* **by** *blast*
qed

theorem *herbrand*:
assumes *unsat*: ∀ *G*. ¬*eval*$_{cs}$ *HFun G Cs*
assumes *finite-cs*: *finite Cs* ∀ *C*∈*Cs*. *finite C*
shows $\exists\, T.$ *closed-tree T Cs*
proof −
from *unsat finite-cs* **obtain** *T* **where** *anybranch T* (λ*b*. *closed-branch b T Cs*) **using** *herbrand′-contra*[*of Cs*] **by** *blast*
then have $\exists\, T.$ *anybranch T* (λ*p*. *falsifies*$_{cs}$ *p Cs*) ∧ *anyinternal T* (λ*p*. ¬ *falsifies*$_{cs}$ *p Cs*)
using *cutoff-branch-internal*[*of T* λ*p*. *falsifies*$_{cs}$ *p Cs*] **by** *blast*
then show *?thesis* **unfolding** *closed-tree-def* **by** *auto*
qed

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*: *finite* $C_1 \wedge$ *finite* C_2
  assumes *apart*: $\mathit{vars}_{ls}\ C_1 \cap \mathit{vars}_{ls}\ C_2 = \{\}$
  assumes *inst*: *instance-of*$_{ls}$ $C_1{'}\ C_1 \wedge$ *instance-of*$_{ls}$ $C_2{'}\ C_2$
  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$_{ls}$ (*resolution* $C_1{'}\ C_2{'}\ L_1{'}\ L_2{'}\ \sigma$) (*resolution* $C_1\ C_2\ L_1\ L_2\ \tau$)
proof −
  — Obtaining the subsets we resolve upon:
  let $?R_1{'} = C_1{'} - L_1{'}$ **and** $?R_2{'} = C_2{'} - L_2{'}$

  from *inst* **obtain** $\gamma\ \mu$ **where** $C_1 \cdot_{ls} \gamma = C_1{'} \wedge C_2 \cdot_{ls} \mu = C_2{'}$
    unfolding *instance-of*$_{ls}$*-def* **by** *auto*
  then obtain η **where** η*-p*: $C_1 \cdot_{ls} \eta = C_1{'} \wedge C_2 \cdot_{ls} \eta = C_2{'}$
    using *apart merge-sub* **by** *force*

  from η*-p* **obtain** L_1 **where** L_1*-p*: $L_1 \subseteq C_1 \wedge L_1 \cdot_{ls} \eta = L_1{'} \wedge (C_1 - L_1) \cdot_{ls} \eta = ?R_1{'}$
    using *appl project-sub* **using** *applicable-def* **by** *metis*
  let $?R_1 = C_1 - L_1$
  from η*-p* **obtain** L_2 **where** L_2*-p*: $L_2 \subseteq C_2 \wedge L_2 \cdot_{ls} \eta = L_2{'} \wedge (C_2 - L_2) \cdot_{ls} \eta = ?R_2{'}$
    using *appl project-sub* **using** *applicable-def* **by** *metis*
  let $?R_2 = C_2 - L_2$

  — Obtaining substitutions:
  from *appl* **have** $\mathit{mgu}_{ls}\ \sigma\ (L_1{'} \cup {L_2{'}}^C)$ **using** *applicable-def* **by** *auto*
  then have $\mathit{mgu}_{ls}\ \sigma\ ((L_1 \cdot_{ls} \eta) \cup (L_2 \cdot_{ls} \eta)^C)$ **using** L_1*-p* L_2*-p* **by** *auto*
  then have $\mathit{mgu}_{ls}\ \sigma\ ((L_1\ \cup {L_2}^C) \cdot_{ls} \eta)$ **using** *compls-subls subls-union* **by** *auto*
  then have *unifier*$_{ls}$ $\sigma\ ((L_1\ \cup {L_2}^C) \cdot_{ls} \eta)$ **using** *mgu*$_{ls}$*-def* **by** *auto*
  then have $\eta\sigma$*uni*: *unifier*$_{ls}$ $(\eta \cdot \sigma)\ (L_1\ \cup {L_2}^C)$
    using *unifier*$_{ls}$*-def composition-conseq2l* **by** *auto*
  then obtain τ **where** τ*-p*: $\mathit{mgu}_{ls}\ \tau\ (L_1\ \cup {L_2}^C)$
   using *unification fin* L_1*-p* L_2*-p* **by** (*meson finite-UnI finite-imageI rev-finite-subset*)
  then obtain φ **where** φ*-p*: $\tau \cdot \varphi = \eta \cdot \sigma$ **using** $\eta\sigma$*uni mgu*$_{ls}$*-def* **by** *auto*

  — Showing that we have the desired resolvent:
  let $?C = ((C_1 - L_1)\ \cup (C_2 - L_2)) \cdot_{ls} \tau$
  have $?C \cdot_{ls} \varphi\ = (?R_1 \cup ?R_2\ ) \cdot_{ls} (\tau \cdot \varphi)$
    using *subls-union composition-conseq2ls* **by** *auto*
  also have ... $= (?R_1 \cup ?R_2\ ) \cdot_{ls} (\eta \cdot \sigma)$ **using** φ*-p* **by** *auto*
  also have ... $= ((?R_1 \cdot_{ls} \eta) \cup (?R_2 \cdot_{ls} \eta)) \cdot_{ls} \sigma$

using *subls-union composition-conseq2ls* **by** *auto*
also have ... = (*?R*$_1$' ∪ *?R*$_2$') ·$_{ls}$ σ **using** η-*p* L_1-*p* L_2-*p* **by** *auto*
finally have *?C* ·$_{ls}$ φ = ((C_1' − L_1') ∪ (C_2' − L_2')) ·$_{ls}$ σ **by** *auto*
then have *ins*: *instance-of*$_{ls}$ (*resolution* C_1' C_2' L_1' L_2' σ) (*resolution* C_1 C_2 L_1 L_2 τ)
using *resolution-def instance-of*$_{ls}$*-def* **by** *metis*

— Showing that the resolution rule is applicable:
have C_1' ≠ {} ∧ C_2' ≠ {} ∧ L_1' ≠ {} ∧ L_2' ≠ {}
using *appl applicable-def* **by** *auto*
then have C_1 ≠ {} ∧ C_2 ≠ {} ∧ L_1 ≠ {} ∧ L_2 ≠ {} **using** η-*p* L_1-*p* L_2-*p* **by** *auto*
then have *appli*: *applicable* C_1 C_2 L_1 L_2 τ
using *apart* L_1-*p* L_2-*p* τ-*p* *applicable-def* **by** *auto*

from *ins appli* **show** *?thesis* **by** *auto*
qed

17 Completeness

lemma *falsifies*$_g$*-empty*:
assumes *falsifies*$_g$ [] *C*
shows *C* = {}
proof −
have ∀ *l* ∈ *C*. *False*
proof
fix *l*
assume *l*∈*C*
then have *falsifies*$_l$ [] *l* **using** *assms* **by** *auto*
then show *False* **unfolding** *falsifies*$_l$*-def* **by** (*cases l*) *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *falsifies*$_{cs}$*-empty*:
assumes *falsifies*$_c$ [] *C*
shows *C* = {}
proof −
from *assms* **obtain** *C'* **where** *C'-p*: *instance-of*$_{ls}$ *C'* *C* ∧ *falsifies*$_g$ [] *C'* **by** *auto*
then have *C'*= {} **using** *falsifies*$_g$*-empty* **by** *auto*
then show *C* = {} **using** *C'-p* **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
qed

lemma *complements-do-not-falsify'*:
assumes *l1C1'*: l_1 ∈ C_1'
assumes l_2*C1'*: l_2 ∈ C_1'
assumes *comp*: l_1 = $l_2{}^c$
assumes *falsif*: *falsifies*$_g$ *G* C_1'

shows *False*
proof (*cases* l_1)
case (*Pos p ts*)
let *?i1* = *nat-of-fatom* (*p*, *ts*)

from *assms* **have** *gr*: $ground_l\ l_1$ **unfolding** $falsifies_l$*-def* **by** *auto*
then have *Neg*: l_2 = *Neg p ts* **using** *comp Pos* **by** (*cases* l_2) *auto*

from *falsif* **have** $falsifies_l\ G\ l_1$ **using** *l1C1'* **by** *auto*
then have *G* ! *?i1* = *False* **using** *l1C1' Pos* **unfolding** $falsifies_l$*-def* **by** (*induction Pos p ts*) *auto*
moreover
let *?i2* = *nat-of-fatom* (*get-atom* l_2)
from *falsif* **have** $falsifies_l\ G\ l_2$ **using** l_2*C1'* **by** *auto*
then have *G* ! *?i2* = (¬*sign* l_2) **unfolding** $falsifies_l$*-def* **by** *meson*
then have *G* ! *?i1* = (¬*sign* l_2) **using** *Pos Neg comp* **by** *simp*
then have *G* ! *?i1* = *True* **using** *Neg* **by** *auto*
ultimately show *?thesis* **by** *auto*
next
case (*Neg p ts*)
let *?i1* = *nat-of-fatom* (*p*,*ts*)

from *assms* **have** *gr*: $ground_l\ l_1$ **unfolding** $falsifies_l$*-def* **by** *auto*
then have *Pos*: l_2 = *Pos p ts* **using** *comp Neg* **by** (*cases* l_2) *auto*

from *falsif* **have** $falsifies_l\ G\ l_1$ **using** *l1C1'* **by** *auto*
then have *G* ! *?i1* = *True* **using** *l1C1' Neg* **unfolding** $falsifies_l$*-def* **by** (*metis get-atom.simps*(*2*) *literal.disc*(*2*))
moreover
let *?i2* = *nat-of-fatom* (*get-atom* l_2)
from *falsif* **have** $falsifies_l\ G\ l_2$ **using** l_2*C1'* **by** *auto*
then have *G* ! *?i2* = (¬*sign* l_2) **unfolding** $falsifies_l$*-def* **by** *meson*
then have *G* ! *?i1* = (¬*sign* l_2) **using** *Pos Neg comp* **by** *simp*
then have *G* ! *?i1* = *False* **using** *Pos* **using** *literal.disc*(*1*) **by** *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *complements-do-not-falsify*:
assumes *l1C1'*: $l_1 \in C_1'$
assumes l_2*C1'*: $l_2 \in C_1'$
assumes *fals*: $falsifies_g\ G\ C_1'$
shows $l_1 \neq {l_2}^c$
using *assms complements-do-not-falsify'* **by** *blast*

lemma *other-falsified*:
assumes *C1'-p*: $ground_{ls}\ C_1' \wedge falsifies_g\ (B@[d])\ C_1'$
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 −
let *?i* = *nat-of-fatom* (*get-atom lo*)
have *ground-l*$_2$: *ground*$_l$ *l* **using** *l-p C1′-p* **by** *auto*
— They are, of course, also ground:
have *ground-lo*: *ground*$_l$ *lo* **using** *C1′-p other* **by** *auto*
from *C1′-p* **have** *falsifies*$_g$ (*B*@[*d*]) (*C*$_1$′ − {*l*}) **by** *auto*
— And indeed, falsified by *B* @ [*d*]:
then have *loB*$_2$: *falsifies*$_l$ (*B*@[*d*]) *lo* **using** *other* **by** *auto*
then have *?i* < *length* (*B* @ [*d*]) **unfolding** *falsifies*$_l$*-def* **by** *meson*
— And they have numbers in the range of *B* @ [*d*], i.e. less than *length B* + *1*:
then have *nat-of-fatom* (*get-atom lo*) < *length B* + *1* **using** *undiag-diag-fatom* **by** (*cases lo*) *auto*
moreover
have *l-lo*: *l*≠*lo* **using** *other* **by** *auto*
— The are not the complement of *l*, since then the clause could not be falsified:
have *lc-lo*: *lo* ≠ *l*^{c} **using** *C1′-p l-p other complements-do-not-falsify*[*of lo C*$_1$′ *l* (*B*@[*d*])] **by** *auto*
from *l-lo lc-lo* **have** *get-atom l* ≠ *get-atom lo* **using** *sign-comp-atom* **by** *metis*
then have *nat-of-fatom* (*get-atom lo*) ≠ *nat-of-fatom* (*get-atom l*)
using *nat-of-fatom-bij ground-lo ground-l*$_2$ *ground*$_l$*-ground-fatom*
unfolding *bij-betw-def inj-on-def* **by** *metis*
— Therefore they have different numbers:
then have *nat-of-fatom* (*get-atom lo*) ≠ *length B* **using** *l-p* **by** *auto*
ultimately
— So their numbers are in the range of *B*:
have *nat-of-fatom* (*get-atom lo*) < *length B* **by** *auto*
— So we did not need the last index of *B* @ [*d*] to falsify them, i.e. *B* suffices:
then show *falsifies*$_l$ *B lo* **using** *loB*$_2$ *shorter-falsifies*$_l$ **by** *blast*
qed

theorem *completeness′*:
assumes *closed-tree T Cs*
assumes ∀ *C*∈*Cs*. *finite C*
shows ∃ *Cs′*. *resolution-deriv Cs Cs′* ∧ {} ∈ *Cs′*
using *assms* **proof** (*induction T arbitrary*: *Cs rule*: *measure-induct-rule*[*of tree-size*])
fix *T* :: *tree*
fix *Cs* :: *fterm clause set*
assume *ih*: ⋀*T′ Cs*. *treesize T′* < *treesize T* ⟹ *closed-tree T′ Cs* ⟹
∀ *C*∈*Cs*. *finite C* ⟹ ∃ *Cs′*. *resolution-deriv Cs Cs′* ∧ {} ∈ *Cs′*
assume *clo*: *closed-tree T Cs*
assume *finite-Cs*: ∀ *C*∈*Cs*. *finite C*
{ — Base case:
assume *treesize T* = *0*
then have *T*=*Leaf* **using** *treesize-Leaf* **by** *auto*
then have *closed-branch* [] *Leaf Cs* **using** *branch-inv-Leaf clo* **unfolding** *closed-tree-def* **by** *auto*
then have *falsifies*$_{cs}$ [] *Cs* **by** *auto*

then have $\{\} \in Cs$ **using** $falsifies_{cs}$*-empty* **by** *auto*
then have $\exists\, Cs'.\ resolution\text{-}deriv\ Cs\ Cs' \wedge \{\} \in Cs'$
unfolding *resolution-deriv-def* **by** *auto*
}
moreover
{ — Induction case:
assume *treesize* $T > 0$
then have $\exists\, l\ r.\ T{=}Branching\ l\ r$ **by** (*cases* T) *auto*

— Finding sibling branches and their corresponding clauses:
then obtain B **where** *b-p*: *internal* $B\ T \wedge branch\ (B@[True])\ T \wedge branch\ (B@[False])\ T$
using *internal-branch*[*of* - [] - T] *Branching-Leaf-Leaf-Tree* **by** *fastforce*
let $?B_1 = B@[True]$
let $?B_2 = B@[False]$

obtain $C_1 o$ **where** $C_1 o$*-p*: $C_1 o \in Cs \wedge falsifies_c\ ?B_1\ C_1 o$ **using** *b-p clo* **unfolding** *closed-tree-def* **by** *metis*
obtain $C_2 o$ **where** $C_2 o$*-p*: $C_2 o \in Cs \wedge falsifies_c\ ?B_2\ C_2 o$ **using** *b-p clo* **unfolding** *closed-tree-def* **by** *metis*

— Standardizing the clauses apart:
let $?C_1 = std_1\ C_1 o$
let $?C_2 = std_2\ C_2 o$
have C_1*-p*: $falsifies_c\ ?B_1\ ?C_1$ **using** std_1*-falsifies* $C_1 o$*-p* **by** *auto*
have C_2*-p*: $falsifies_c\ ?B_2\ ?C_2$ **using** std_2*-falsifies* $C_2 o$*-p* **by** *auto*

have *fin*: *finite* $?C_1 \wedge$ *finite* $?C_2$ **using** $C_1 o$*-p* $C_2 o$*-p finite-Cs* **by** *auto*

— We go down to the ground world.
— Finding the falsifying ground instance $C_1{'}$ of $std_1\ C_1 o$, and proving properties about it:

— $C_1{'}$ is falsified by B @ [*True*]:
from C_1*-p* **obtain** $C_1{'}$ **where** $C_1{'}$*-p*: $ground_{ls}\ C_1{'} \wedge instance\text{-}of_{ls}\ C_1{'}\ ?C_1 \wedge falsifies_g\ ?B_1\ C_1{'}$ **by** *metis*

have $\neg falsifies_c\ B\ C_1 o$ **using** $C_1 o$*-p b-p clo* **unfolding** *closed-tree-def* **by** *metis*
then have $\neg falsifies_c\ B\ ?C_1$ **using** std_1*-falsifies* **using** *prod.exhaust-sel* **by** *blast*
— $C_1{'}$ is not falsified by B:
then have *l-B*: $\neg falsifies_g\ B\ C_1{'}$ **using** $C_1{'}$*-p* **by** *auto*

— $C_1{'}$ contains a literal l_1 that is falsified by B @ [*True*], but not B:
from $C_1{'}$*-p l-B* **obtain** l_1 **where** l_1*-p*: $l_1 \in C_1{'} \wedge falsifies_l\ (B@[True])\ l_1 \wedge \neg(falsifies_l\ B\ l_1)$ **by** *auto*
let $?i = nat\text{-}of\text{-}fatom\ (get\text{-}atom\ l_1)$

— l_1 is of course ground:

have *ground-l_1*: $ground_l\ l_1$ **using** *C_1'-p l_1-p* **by** *auto*

from *l_1-p* **have** $\neg(?i < length\ B \wedge B\ !\ ?i = (\neg sign\ l_1))$ **using** *ground-l_1* **unfolding** *$falsifies_l$-def* **by** *meson*
then have $\neg(?i < length\ B \wedge (B@[True])\ !\ ?i = (\neg sign\ l_1))$ **by** (*metis nth-append*) — Not falsified by *B*.
moreover
from *l_1-p* **have** $?i < length\ (B\ @\ [True]) \wedge (B\ @\ [True])\ !\ ?i = (\neg sign\ l_1)$ **unfolding** *$falsifies_l$-def* **by** *meson*
ultimately
have *l_1-sign-no*: $?i = length\ B \wedge (B\ @\ [True])\ !\ ?i = (\neg sign\ l_1)$ **by** *auto*

— l_1 is negative:
from *l_1-sign-no* **have** *l_1-sign*: $sign\ l_1 = False$ **by** *auto*
from *l_1-sign-no* **have** *l_1-no*: $nat\text{-}of\text{-}fatom\ (get\text{-}atom\ l_1) = length\ B$ **by** *auto*

— All the other literals in C_1' must be falsified by B, since they are falsified by $B\ @\ [True]$, but not l_1.
from *C_1'-p l_1-no l_1-p* **have** *B-C_1'l_1*: $falsifies_g\ B\ (C_1' - \{l_1\})$
using *other-falsified* **by** *blast*

— We do the same exercise for $std_2\ C_2o$, C_2', $B\ @\ [False]$, l_2:
from *C_2-p* **obtain** *C_2'* **where** *C_2'-p*: $ground_{ls}\ C_2' \wedge instance\text{-}of_{ls}\ C_2'\ ?C_2 \wedge falsifies_g\ ?B_2\ C_2'$ **by** *metis*

have $\neg falsifies_c\ B\ C_2o$ **using** *C_2o-p b-p clo* **unfolding** *closed-tree-def* **by** *metis*
then have $\neg falsifies_c\ B\ ?C_2$ **using** *std_2-falsifies* **using** *prod.exhaust-sel* **by** *blast*
then have *l-B*: $\neg falsifies_g\ B\ C_2'$ **using** *C_2'-p* **by** *auto*

— C_2' contains a literal l_2 that is falsified by $B\ @\ [False]$, but not B:
from *C_2'-p l-B* **obtain** *l_2* **where** *l_2-p*: $l_2 \in C_2' \wedge falsifies_l\ (B@[False])\ l_2 \wedge \neg falsifies_l\ B\ l_2$ **by** *auto*
let $?i = nat\text{-}of\text{-}fatom\ (get\text{-}atom\ l_2)$

have *ground-l_2*: $ground_l\ l_2$ **using** *C_2'-p l_2-p* **by** *auto*

from *l_2-p* **have** $\neg(?i < length\ B \wedge B\ !\ ?i = (\neg sign\ l_2))$ **using** *ground-l_2* **unfolding** *$falsifies_l$-def* **by** *meson*
then have $\neg(?i < length\ B \wedge (B@[False])\ !\ ?i = (\neg sign\ l_2))$ **by** (*metis nth-append*) — Not falsified by *B*.
moreover
from *l_2-p* **have** $?i < length\ (B\ @\ [False]) \wedge (B\ @\ [False])\ !\ ?i = (\neg sign\ l_2)$ **unfolding** *$falsifies_l$-def* **by** *meson*
ultimately
have *l_2-sign-no*: $?i = length\ B \wedge (B\ @\ [False])\ !\ ?i = (\neg sign\ l_2)$ **by** *auto*

— l_2 is negative:
from *l_2-sign-no* **have** *l_2-sign*: $sign\ l_2 = True$ **by** *auto*

from l_2*-sign-no* **have** l_2*-no*: *nat-of-fatom* (*get-atom* l_2) = *length* B **by** *auto*

— All the other literals in $C_2{'}$ must be falsified by B, since they are falsified by B @ [*False*], but not l_2.
from $C_2{'}$*-p* l_2*-no* l_2*-p* **have** B-$C_2{'}l_2$: *falsifies*$_g$ B ($C_2{'}$ − {l_2})
using *other-falsified* **by** *blast*

— Proving some properties about $C_1{'}$ and $C_2{'}$, l_1 and l_2, as well as the resolvent of $C_1{'}$ and $C_2{'}$:
have $l_2 cisl_1$: $l_2{}^c = l_1$
proof −
from l_1*-no* l_2*-no ground-*l_1 *ground-*l_2 **have** *get-atom* l_1 = *get-atom* l_2
using *nat-of-fatom-bij ground*$_l$*-ground-fatom*
unfolding *bij-betw-def inj-on-def* **by** *metis*
then show $l_2{}^c = l_1$ **using** l_1*-sign* l_2*-sign* **using** *sign-comp-atom* **by** *metis*
qed

have *applicable* $C_1{'}$ $C_2{'}$ {l_1} {l_2} *Resolution.*ε **unfolding** *applicable-def*
using l_1*-p* l_2*-p* $C_1{'}$*-p ground*$_{ls}$*-vars*$_{ls}$ $l_2 cisl_1$ *empty-comp2* **unfolding** *mgu*$_{ls}$*-def unifier*$_{ls}$*-def* **by** *auto*
— Lifting to get a resolvent of *std*$_1$ $C_1 o$ and *std*$_2$ $C_2 o$:
then obtain L_1 L_2 τ **where** $L_1 L_2 \tau$*-p*: *applicable ?*C_1 *?*C_2 L_1 L_2 τ ∧ *instance-of*$_{ls}$ (*resolution* $C_1{'}$ $C_2{'}$ {l_1} {l_2} *Resolution.*ε) (*resolution ?*C_1 *?*C_2 L_1 L_2 τ)
using *std-apart-apart* $C_1{'}$*-p* $C_2{'}$*-p lifting*[*of ?*C_1 *?*C_2 $C_1{'}$ $C_2{'}$ {l_1} {l_2} *Resolution.*ε] *fin* **by** *auto*

— Defining the clause to be derived, the new clausal form and the new tree:
— We name the resolvent C.
obtain C **where** C*-p*: C = *resolution ?*C_1 *?*C_2 L_1 L_2 τ **by** *auto*
obtain *CsNext* **where** *CsNext-p*: *CsNext* = Cs ∪ {*?*C_1, *?*C_2, C} **by** *auto*
obtain T'' **where** T''*-p*: T'' = *delete* B T **by** *auto*
— Here we delete the two branch children B @ [*True*] and B @ [*False*] of B.

— Our new clause is falsified by the branch B of our new tree:
have *falsifies*$_g$ B (($C_1{'}$ − {l_1}) ∪ ($C_2{'}$ − {l_2})) **using** B-$C_1{'}l_1$ B-$C_2{'}l_2$ **by** *cases auto*
then have *falsifies*$_g$ B (*resolution* $C_1{'}$ $C_2{'}$ {l_1} {l_2} *Resolution.*ε) **unfolding** *resolution-def empty-subls* **by** *auto*
then have *falsifies-C*: *falsifies*$_c$ B C **using** C*-p* $L_1 L_2 \tau$*-p* **by** *auto*

have T''*-smaller*: *treesize* T'' < *treesize* T **using** *treezise-delete* T''*-p* *b-p* **by** *auto*
have T''*-bran*: *anybranch* T'' (λ*b*. *closed-branch b* T'' *CsNext*)
proof (*rule allI*; *rule impI*)
fix *b*
assume *br*: *branch b* T''
from *br* **have** *b* = B ∨ *branch b* T **using** *branch-delete* T''*-p* **by** *auto*

then show *closed-branch b T'' CsNext*
proof
assume *b=B*
then show *closed-branch b T'' CsNext* **using** *falsifies-C br CsNext-p* **by** *auto*
next
assume *branch b T*
then show *closed-branch b T'' CsNext* **using** *clo br T''-p CsNext-p* **unfolding** *closed-tree-def* **by** *auto*
qed
qed
then have *T''-bran2*: *anybranch T''* (λ*b*. *falsifies*$_{cs}$ *b CsNext*) **by** *auto*

— We cut the tree even smaller to ensure only the branches are falsified, i.e. it is a closed tree:
obtain *T'* **where** *T'-p*: *T'* = *cutoff* (λ*G*. *falsifies*$_{cs}$ *G CsNext*) [] *T''* **by** *auto*
have *T'-smaller*: *treesize T'* < *treesize T* **using** *treesize-cutoff*[*of* λ*G*. *falsifies*$_{cs}$ *G CsNext* [] *T''*] *T''-smaller* **unfolding** *T'-p* **by** *auto*

from *T''-bran2* **have** *anybranch T'* (λ*b*. *falsifies*$_{cs}$ *b CsNext*) **using** *cut-off-branch*[*of T''* λ*b*. *falsifies*$_{cs}$ *b CsNext*] *T'-p* **by** *auto*
then have *T'-bran*: *anybranch T'* (λ*b*. *closed-branch b T' CsNext*) **by** *auto*
have *T'-intr*: *anyinternal T'* (λ*p*. ¬*falsifies*$_{cs}$ *p CsNext*) **using** *T'-p cut-off-internal*[*of T''* λ*b*. *falsifies*$_{cs}$ *b CsNext*] *T''-bran2* **by** *blast*
have *T'-closed*: *closed-tree T' CsNext* **using** *T'-bran T'-intr* **unfolding** *closed-tree-def* **by** *auto*
have *finite-CsNext*: ∀ *C*∈*CsNext*. *finite C* **unfolding** *CsNext-p C-p resolution-def* **using** *finite-Cs fin* **by** *auto*

— By induction hypothesis we get a resolution derivation of {} from our new clausal form:
from *T'-smaller T'-closed* **have** ∃ *Cs''*. *resolution-deriv CsNext Cs''* ∧ {} ∈ *Cs''* **using** *ih*[*of T' CsNext*] *finite-CsNext* **by** *blast*
then obtain *Cs''* **where** *Cs''-p*: *resolution-deriv CsNext Cs''* ∧ {} ∈ *Cs''* **by** *auto*
moreover
{ — Proving that we can actually derive the new clausal form:
have *resolution-step Cs* (*Cs* ∪ {*?C*$_1$}) **using** *std*$_1$*-renames standardize-apart C*$_1$*o-p* **by** (*metis Un-insert-right*)
moreover
have *resolution-step* (*Cs* ∪ {*?C*$_1$}) (*Cs* ∪ {*?C*$_1$} ∪ {*?C*$_2$}) **using** *std*$_2$*-renames*[*of C*$_2$*o*] *standardize-apart*[*of C*$_2$*o* - *?C*$_2$] *C*$_2$*o-p* **by** *auto*
then have *resolution-step* (*Cs* ∪ {*?C*$_1$}) (*Cs* ∪ {*?C*$_1$,*?C*$_2$}) **by** (*simp add*: *insert-commute*)
moreover
then have *resolution-step* (*Cs* ∪ {*?C*$_1$,*?C*$_2$}) (*Cs* ∪ {*?C*$_1$,*?C*$_2$} ∪ {*C*})
using *L*$_1$*L*$_2$τ*-p resolution-rule*[*of ?C*$_1$ *Cs* ∪ {*?C*$_1$,*?C*$_2$} *?C*$_2$ *L*$_1$ *L*$_2$ τ] **using** *C-p* **by** *auto*
then have *resolution-step* (*Cs* ∪ {*?C*$_1$,*?C*$_2$}) *CsNext* **using** *CsNext-p* **by**

(*simp add*: *Un-commute*)
    ultimately
    have *resolution-deriv Cs CsNext* **unfolding** *resolution-deriv-def* **by** *auto*
  }
  — Combining the two derivations, we get the desired derivation from *Cs* of {}:
  ultimately have *resolution-deriv Cs Cs''* **unfolding** *resolution-deriv-def* **by** *auto*
  then have $\exists$ *Cs'. resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'* **using** *Cs''-p* **by** *auto*
 }
 ultimately show $\exists$ *Cs'. resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'* **by** *auto*
qed

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 −
 from *unsat* **have** $\forall$ (*G*::*hterm pred-denot*) . $\neg$*eval*$_{cs}$ *HFun G Cs* **by** *auto*
 then obtain *T* **where** *closed-tree T Cs* **using** *herbrand assms* **by** *blast*
 then show $\exists$ *Cs'. resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'* **using** *completeness' assms*
by *auto*
qed

definition *E-conv* :: ('*a* $\Rightarrow$ '*b*) $\Rightarrow$ '*a var-denot* $\Rightarrow$ '*b var-denot* **where**
 E-conv b-of-a E $\equiv$ λ*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 $\equiv$ λ*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 $\equiv$ λ*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 (*induction t*)
 case (*Fun f ts*)
 then have *map* (*inv b-of-a* $\circ$ *eval*$_t$ (*E-conv b-of-a E*) (*F-conv b-of-a F*)) *ts* =
eval$_{ts}$ *E F ts*
  unfolding *E-conv-def F-conv-def*
  using *assms bij-is-inj* **by** *fastforce*
 then have *b-of-a* (*F f* (*map* (*inv b-of-a* $\circ$ *eval*$_t$ (*E-conv b-of-a E*) ((*F-conv b-of-a*
F))) *ts*)) = *b-of-a* (*F f* (*eval*$_{ts}$ *E F ts*)) **by** *metis*
 then show *?case* **using** *assms* **unfolding** *E-conv-def F-conv-def* **by** *auto*
next
 case (*Var x*)
 then show *?case* **using** *assms* **unfolding** *E-conv-def* **by** *auto*
qed

lemma $eval_{ts}$-*bij*:
assumes *bij* (*b-of-a*::'*a* ⇒ '*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*)
using *assms* **using** $eval_t$-*bij*
proof −
have *map* (*inv b-of-a* ∘ $eval_t$ (*E-conv b-of-a E*) (*F-conv b-of-a F*)) *ts* = $eval_{ts}$ *E F ts*
using $eval_t$-*bij assms bij-is-inj* **by** *fastforce*
then show *?thesis*
by (*metis* (*no-types*) *G-conv-def map-map*)
qed

lemma $eval_l$-*bij*:
assumes *bij* (*b-of-a*::'*a* ⇒ '*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*
using *assms* $eval_{ts}$-*bij*
proof (*cases l*)
case (*Pos p ts*)
then show *?thesis*
by (*simp add*: $eval_{ts}$-*bij assms*)
next
case (*Neg p ts*)
then show *?thesis*
by (*simp add*: $eval_{ts}$-*bij assms*)
qed

lemma $eval_c$-*bij*:
assumes *bij* (*b-of-a*::'*a* ⇒ '*b*)
shows $eval_c$ (*F-conv b-of-a F*) (*G-conv b-of-a G*) *C* = $eval_c$ *F G C*
proof −
{
fix *E* :: *char list* ⇒ '*b*
assume *bij-b-of-a*: *bij b-of-a*
assume *C-sat*: ∀ *E* :: *char list* ⇒ '*a*. ∃ *l*∈*C*. $eval_l$ *E F G l*
have *E-p*: *E* = *E-conv b-of-a* (*E-conv* (*inv b-of-a*) *E*)
unfolding *E-conv-def* **using** *bij-b-of-a*
using *bij-betw-inv-into-right* **by** *fastforce*
have ∃ *l*∈*C*. $eval_l$ (*E-conv b-of-a* (*E-conv* (*inv b-of-a*) *E*)) (*F-conv b-of-a F*) (*G-conv b-of-a G*) *l*
using $eval_l$-*bij bij-b-of-a C-sat* **by** *blast*
then have ∃ *l*∈*C*. $eval_l$ *E* (*F-conv b-of-a F*) (*G-conv b-of-a G*) *l* **using** *E-p* **by** *auto*
}
then show *?thesis*
by (*meson* $eval_l$-*bij assms* $eval_c$-*def*)
qed

lemma *eval$_{cs}$-bij*:
assumes *bij* (*b-of-a*::*'a* ⇒ *'b*)
shows *eval$_{cs}$* (*F-conv b-of-a F*) (*G-conv b-of-a G*) *Cs* ⟷ *eval$_{cs}$ F G Cs*
by (*meson eval$_c$-bij assms eval$_{cs}$-def*)

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 −
let *?S* = *UNIV* :: ((*'a*::*countable*)) *set*
have *countable ?S* **by** *auto*
moreover
have *infinite ?S* **using** *inf-a-uni* **by** *auto*
ultimately
obtain *nat-of-a* **where** *QWER*: *bij* (*nat-of-a* :: *'a* ⇒ *nat*) **using** *countableE-infinite*[*of ?S*] **by** *blast*

let *?T* = *UNIV* :: ((*'b*::*countable*)) *set*
have *countable ?T* **by** *auto*
moreover
have *infinite ?T* **using** *inf-b-uni* **by** *auto*
ultimately
obtain *nat-of-b* **where** *TYUI*: *bij* (*nat-of-b* :: *'b* ⇒ *nat*) **using** *countableE-infinite*[*of ?T*] **by** *blast*

let *?b-of-a* = λ*a*. (*inv nat-of-b*) (*nat-of-a a*)

have *bij-nat-of-b*: ∀ *n*. *nat-of-b* (*inv nat-of-b n*) = *n*
using *TYUI bij-betw-inv-into-right* **by** *fastforce*
have ∀ *a*. *inv nat-of-a* (*nat-of-a a*) = *a*
by (*meson QWER UNIV-I bij-betw-inv-into-left*)
then have *inj* (λ*a*. *inv nat-of-b* (*nat-of-a a*))
using *bij-nat-of-b injI* **by** (*metis* (*no-types*))
moreover
have *range* (λ*a*. *inv nat-of-b* (*nat-of-a a*)) = *UNIV*
by (*metis QWER TYUI bij-def image-image inj-imp-surj-inv*)
ultimately
have *bij ?b-of-a*
unfolding *bij-def* **by** *auto*

then show *?thesis* **by** *auto*
qed

lemma *infinite-hterms*: *infinite* (*UNIV* :: *hterm set*)
proof −
let *?diago* = λ*n*. *HFun* (*string-of-nat n*) []
let *?undiago* = λ*a*. *nat-of-string* (*case a of HFun f ts* ⇒ *f*)

have $\forall n.$ *?undiago* (*?diago* n) $= n$ **using** *nat-of-string-string-of-nat* **by** *auto*
moreover
have $\forall n.$ *?diago* $n \in$ *UNIV* **by** *auto*
ultimately show *infinite* (*UNIV* :: *hterm set*) **using** *infinity*[*of ?undiago ?diago UNIV*] **by** *simp*
qed

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 fun-denot*) (*G*::'*u pred-denot*). $\neg$*eval*$_{cs}$ *F G Cs*
shows $\exists$ *Cs'*. *resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'*
proof −
have $\forall$(*F*::*hterm fun-denot*) (*G*::*hterm pred-denot*) . $\neg$*eval*$_{cs}$ *F G Cs*
proof (*rule*; *rule*)
fix *F* :: *hterm fun-denot*
fix *G* :: *hterm pred-denot*

obtain *u-of-hterm* :: *hterm* $\Rightarrow$ '*u* **where** *p-u-of-hterm*: *bij u-of-hterm*
using *countably-inf-bij inf-uni infinite-hterms* **by** *auto*

let *?F* = *F-conv u-of-hterm F*
let *?G* = *G-conv u-of-hterm G*

have $\neg$ *eval*$_{cs}$ *?F ?G Cs* **using** *unsat* **by** *auto*
then show $\neg$ *eval*$_{cs}$ *F G Cs* **using** *eval*$_{cs}$*-bij* **using** *p-u-of-hterm* **by** *auto*
qed
then show $\exists$ *Cs'*. *resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'* **using** *finite-cs completeness* **by** *auto*
qed

theorem *completeness-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'*
using *assms completeness-countable* **by** *blast*

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
  by auto
lemma eval_t E_nat F_nat (Fun ''one'' []) = 1
  by auto
lemma eval_t E_nat F_nat (Fun ''mul'' [Var ''y'',Var ''y'']) = 25
  by auto
lemma
  eval_t E_nat F_nat (Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''], Fun ''one'' []]) =
26
  by auto

lemma eval_l E_nat F_nat G_nat (Pos ''greater'' [Var ''x'', Var ''y'']) = True
  by auto
lemma eval_l E_nat F_nat G_nat (Neg ''less'' [Var ''x'', Var ''y'']) = True
  by auto
lemma eval_l E_nat F_nat G_nat (Pos ''less'' [Var ''x'', Var ''y'']) = False
  by auto

lemma eval_l E_nat F_nat G_nat
      (Pos ''equals''
        [Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''],Fun ''one'' []]
```

```
      ,Var ''x'']
    ) = True
  by auto

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 ε L
  shows mgu_ls ε L
using assms unfolding unifier_ls-def mgu_ls-def apply auto
apply (rule-tac x=u in exI)
using empty-comp1 empty-comp2 apply auto
done

theorem unifier-single: unifier_ls σ {l}
unfolding unifier_ls-def by auto

theorem resolution-rule':
  assumes C_1 ∈ Cs
  assumes C_2 ∈ Cs
  assumes applicable C_1 C_2 L_1 L_2 σ
  assumes C = {resolution C_1 C_2 L_1 L_2 σ}
  shows resolution-step Cs (Cs ∪ C)
  using assms resolution-rule by auto

lemma resolution-example1:
   resolution-deriv {{NP,PQ},{NQ},{PP,PQ}}
                    {{NP,PQ},{NQ},{PP,PQ},{NP},{PP},{}}
proof −
  have resolution-step
        {{NP,PQ},{NQ},{PP,PQ}}
       ({{NP,PQ},{NQ},{PP,PQ}} ∪ {{NP}})
    apply (rule resolution-rule'[of {NP,PQ} - {NQ} {PQ} {NQ} ε])
      unfolding applicable-def vars_ls-def  vars_l-def
           NQ-def NP-def PQ-def PP-def resolution-def
      using unifier-single empty-mgu using empty-subls
      apply auto
    done
  then have resolution-step
```

```
        {{NP,PQ},{NQ},{PP,PQ}}
       ({{NP,PQ},{NQ},{PP,PQ},{NP}})
  by (simp add: insert-commute)
 moreover
 have resolution-step
       {{NP,PQ},{NQ},{PP,PQ},{NP}}
      ({{NP,PQ},{NQ},{PP,PQ},{NP}} ∪ {{PP}})
  apply (rule resolution-rule'[of {NQ} - {PP,PQ} {NQ} {PQ} ε])
    unfolding applicable-def vars_ls-def  vars_l-def
          NQ-def NP-def PQ-def PP-def resolution-def
    using unifier-single empty-mgu empty-subls apply auto
  done
 then have resolution-step
       {{NP,PQ},{NQ},{PP,PQ},{NP}}
      ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP}})
  by (simp add: insert-commute)
 moreover
 have resolution-step
       {{NP,PQ},{NQ},{PP,PQ},{NP},{PP}}
      ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP}} ∪ {{}})
  apply (rule resolution-rule'[of {NP} - {PP} {NP} {PP} ε])
    unfolding applicable-def vars_ls-def  vars_l-def
          NQ-def NP-def PQ-def PP-def resolution-def
    using unifier-single empty-mgu apply auto
  done
 then have resolution-step
       {{NP,PQ},{NQ},{PP,PQ},{NP},{PP}}
      ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP},{}})
  by (simp add: insert-commute)
 ultimately
 have resolution-deriv {{NP,PQ},{NQ},{PP,PQ}}
                       {{NP,PQ},{NQ},{PP,PQ},{NP},{PP},{}}
  unfolding resolution-deriv-def by auto
 then show ?thesis by auto
qed

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 = (λx. if x = ''x'' then Fun ''a'' [] else Var x)

lemma mguPaaPax-mgu: mgu_ls mguPaaPax {Paa,Pax}
proof −
  let ?σ = λx. if x = ''x'' then Fun ''a'' [] else Var x
  have a: unifier_ls (λx. if x = ''x'' then Fun ''a'' [] else Var x) {Paa,Pax} unfolding Paa-def Pax-def unifier_ls-def by auto
  have b: ∀ u. unifier_ls u {Paa,Pax} ⟶ (∃ i. u = ?σ · i)
    proof (rule;rule)
      fix u
      assume unifier_ls u {Paa,Pax}
     then have uuu: u ''x'' = Fun ''a'' [] unfolding unifier_ls-def Paa-def Pax-def by auto
      have ?σ · u = u
        proof
          fix x
          {
            assume x=''x''
            moreover
            have (?σ · u) ''x'' =  Fun ''a'' [] unfolding composition-def by auto
            ultimately have (?σ · u) x = u x using uuu by auto
          }
          moreover
          {
            assume x≠''x''
            then have (?σ · u) x = (ε x) ·_t u unfolding composition-def by auto
            then have (?σ · u) x = u x by auto
          }
          ultimately show (?σ · u) x = u x by auto
        qed
      then have ∃ i. ?σ · i = u by auto
      then show ∃ i. u = ?σ · i by auto
    qed
   from a b show ?thesis unfolding mgu_ls-def unfolding mguPaaPax-def by auto
qed
```

lemma *resolution-example2*:
  resolution-deriv {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
                    {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*},{}}
proof −
  have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}} ∪ {{*Na*,*Pb*}})
   apply (*rule resolution-rule'*[*of* {*Pax*} - {*Na*,*Pb*,*Naa*} {*Pax*} {*Naa*} *mguPaaPax*
])
      using *mguPaaPax-mgu* **unfolding** *applicable-def vars$_{ls}$-def vars$_{l}$-def*
          Nb-def Na-def Pax-def Pa-def Pb-def Naa-def Paa-def mguPaaPax-def
resolution-def
      apply *auto*
     apply (*rule-tac x*=*Na* **in** *image-eqI*)
      unfolding *Na-def* **apply** *auto*
    apply (*rule-tac x*=*Pb* **in** *image-eqI*)
     unfolding *Pb-def* **apply** *auto*
    done
  then have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*}})
    by (*simp add*: *insert-commute*)
  moreover
  have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*}} ∪ {{*Na*}})
    apply (*rule resolution-rule'*[*of* {*Nb*,*Na*} - {*Na*,*Pb*} {*Nb*} {*Pb*} *ε*])
      unfolding *applicable-def vars$_{ls}$-def vars$_{l}$-def*
          Pb-def Nb-def Na-def PP-def resolution-def
      using *unifier-single empty-mgu* **apply** *auto*
    done
  then have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*}})
    by (*simp add*: *insert-commute*)
  moreover
  have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*}} ∪ {{}})
    apply (*rule resolution-rule'*[*of* {*Na*} - {*Pa*} {*Na*} {*Pa*} *ε*])
      unfolding *applicable-def vars$_{ls}$-def vars$_{l}$-def*
          Pa-def Nb-def Na-def PP-def resolution-def
      using *unifier-single empty-mgu* **apply** *auto*
    done
  then have *resolution-step*
      {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*}}
      ({{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*},{}})
    by (*simp add*: *insert-commute*)
  ultimately

have *resolution-deriv* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
{{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*},{}}
unfolding *resolution-deriv-def* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *resolution-example1-sem*: $\neg eval_{cs}$ *F G* {{*NP*, *PQ*}, {*NQ*}, {*PP*, *PQ*}}
using *resolution-example1 derivation-sound-refute* **by** *auto*

lemma *resolution-example2-sem*: $\neg eval_{cs}$ *F G* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
using *resolution-example2 derivation-sound-refute* **by** *auto*

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 (*induction rule*: *finite.induct*)
case (*emptyI*)
have *set* [] = {} **by** *auto*
then show *?case* **unfolding** *set-to-list-def inv-into-def* **by** *auto*
next
case (*insertI A a*)
then have *set* (*a*#*set-to-list A*) = *insert a A* **by** *auto*
then show *?case* **unfolding** *set-to-list-def inv-into-def* **by** (*metis* (*mono-tags*, *lifting*) *UNIV-I someI*)
qed

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*
by (*induction t*) (*auto simp add*: *map-idI*)

lemma *fterm-to-iterm-cancel*[*simp*]: *fterm-to-iterm* (*iterm-to-fterm t*) = *t*
by (*induction t*) (*auto simp add*: *map-idI*)

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*)))
by (*induction t1*) *auto*

lemma *unifiert-unifiers*:
assumes *unifier*$_{ts}$ σ *ts*
shows *fsub-to-isub* σ ∈ *unifiers* (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
proof −
have ∀ *t1* ∈ *fterm-to-iterm* ‘ *ts*. ∀ *t2* ∈ *fterm-to-iterm* ‘ *ts*. *t1* · (*fsub-to-isub* σ) = *t2* · (*fsub-to-isub* σ)
proof (*rule ballI*;*rule ballI*)
fix *t1 t2*
assume *t1-p*: *t1* ∈ *fterm-to-iterm* ‘ *ts* **assume** *t2-p*: *t2* ∈ *fterm-to-iterm* ‘ *ts*
from *t1-p t2-p* **have** *iterm-to-fterm t1* ∈ *ts* ∧ *iterm-to-fterm t2* ∈ *ts* **by** *auto*
then have (*iterm-to-fterm t1*) ·$_t$ σ = (*iterm-to-fterm t2*) ·$_t$ σ **using** *assms* **unfolding** *unifier*$_{ts}$*-def* **by** *auto*
then have *iterm-to-fterm* (*t1* · *fsub-to-isub* σ) = *iterm-to-fterm* (*t2* · *fsub-to-isub* σ) **using** *iterm-to-fterm-subt* **by** *auto*
then have *fterm-to-iterm* (*iterm-to-fterm* (*t1* · *fsub-to-isub* σ)) = *fterm-to-iterm* (*iterm-to-fterm* (*t2* · *fsub-to-isub* σ)) **by** *auto*
then show *t1* · *fsub-to-isub* σ = *t2* · *fsub-to-isub* σ **using** *fterm-to-iterm-cancel* **by** *auto*
qed
then have ∀ *p*∈*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*. *fst p* · *fsub-to-isub* σ = *snd p* · *fsub-to-isub* σ **by** (*metis mem-Times-iff*)
then show *?thesis* **unfolding** *unifiers-def* **by** *blast*
qed

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 σ ∈ *unifiers* (*set E*)
shows ∃ ϑ. *is-imgu* ϑ (*set E*)
proof −
from *assms* **have** ∃ *cs*. *unify E* [] = *Some cs* **using** *unify-complete* **by** *auto*
then show *?thesis* **using** *unify-sound* **by** *auto*
qed

lemma *fterm-to-iterm-subst*: (*fterm-to-iterm t1*) · σ =*fterm-to-iterm* (*t1* ·$_t$ *isub-to-fsub* σ)
by (*induction t1*) *auto*

lemma *unifiers-unifiert*:
assumes $\sigma \in$ *unifiers* (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
shows $unifier_{ts}$ (*isub-to-fsub* σ) *ts*
proof (*cases ts*={})
assume *ts* = {}
then show $unifier_{ts}$ (*isub-to-fsub* σ) *ts* **unfolding** $unifier_{ts}$*-def* **by** *auto*
next
assume *ts* ≠ {}
then obtain *t′* **where** *t′-p*: *t′* ∈ *ts* **by** *auto*

have ∀ *t*$_1$∈*ts*. ∀ *t*$_2$∈*ts*. *t*$_1$ ·$_t$ *isub-to-fsub* σ = *t*$_2$ ·$_t$ *isub-to-fsub* σ
proof (*rule ballI* ; *rule ballI*)
fix *t*$_1$ *t*$_2$
assume *t*$_1$ ∈ *ts t*$_2$ ∈ *ts*
then have *fterm-to-iterm t*$_1$ ∈ *fterm-to-iterm* ‘ *ts fterm-to-iterm t*$_2$ ∈ *fterm-to-iterm* ‘ *ts* **by** *auto*
then have (*fterm-to-iterm t*$_1$, *fterm-to-iterm t*$_2$) ∈ (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*) **by** *auto*
then have (*fterm-to-iterm t*$_1$) · σ = (*fterm-to-iterm t*$_2$) · σ **using** *assms* **unfolding** *unifiers-def*
by (*metis* (*no-types*, *lifting*) *assms fst-conv in-unifiersE snd-conv*)
then have *fterm-to-iterm* (*t*$_1$ ·$_t$ *isub-to-fsub* σ) = *fterm-to-iterm* (*t*$_2$ ·$_t$ *isub-to-fsub* σ) **using** *fterm-to-iterm-subst* **by** *auto*
then have *iterm-to-fterm* (*fterm-to-iterm* (*t*$_1$ ·$_t$ (*isub-to-fsub* σ))) = *iterm-to-fterm* (*fterm-to-iterm* (*t*$_2$ ·$_t$ *isub-to-fsub* σ)) **by** *auto*
then show *t*$_1$ ·$_t$ *isub-to-fsub* σ = *t*$_2$ ·$_t$ *isub-to-fsub* σ **by** *auto*
qed
then have ∀ *t*$_2$∈*ts*. *t′* ·$_t$ *isub-to-fsub* σ = *t*$_2$ ·$_t$ *isub-to-fsub* σ **using** *t′-p* **by** *blast*

then show $unifier_{ts}$ (*isub-to-fsub* σ) *ts* **unfolding** $unifier_{ts}$*-def* **by** *metis*
qed

lemma *icomp-fcomp*: $\vartheta \circ_s i$ = *fsub-to-isub* (*isub-to-fsub* ϑ · *isub-to-fsub i*)
unfolding *composition-def subst-compose-def*
proof
fix *x*
show $\vartheta\ x \cdot i$ = *fterm-to-iterm* (*iterm-to-fterm* (ϑ *x*) ·$_t$ (λ*x*. *iterm-to-fterm* (*i x*)))
using *iterm-to-fterm-subt* **by** *auto*
qed

lemma $is\text{-}mgu\text{-}mgu_{ts}$:
assumes *finite ts*
assumes *is-imgu* ϑ (*fterm-to-iterm* ‘ *ts* × *fterm-to-iterm* ‘ *ts*)
shows mgu_{ts} (*isub-to-fsub* ϑ) *ts*
proof −
from *assms* **have** $unifier_{ts}$ (*isub-to-fsub* ϑ) *ts* **unfolding** *is-imgu-def* **using** *unifiers-unifiert* **by** *auto*

moreover have $\forall u.\ unifier_{ts}\ u\ ts \longrightarrow (\exists i.\ u = (isub\text{-}to\text{-}fsub\ \vartheta) \cdot i)$
proof (*rule allI*; *rule impI*)
fix u
assume $unifier_{ts}\ u\ ts$
then have *fsub-to-isub* $u \in$ *unifiers* (*fterm-to-iterm* ‘ *ts* $\times$ *fterm-to-iterm* ‘ *ts*) **using** *unifiert-unifiers* **by** *auto*
then have $\exists i.$ *fsub-to-isub* $u = \vartheta \circ_s i$ **using** *assms* **unfolding** *is-imgu-def* **by** *auto*
then obtain i **where** *fsub-to-isub* $u = \vartheta \circ_s i$ **by** *auto*
then have *fsub-to-isub* $u =$ *fsub-to-isub* (*isub-to-fsub* $\vartheta \cdot$ *isub-to-fsub* i) **using** *icomp-fcomp* **by** *auto*
then have *isub-to-fsub* (*fsub-to-isub* u) = *isub-to-fsub* (*fsub-to-isub* (*isub-to-fsub* $\vartheta \cdot$ *isub-to-fsub* i)) **by** *metis*
then have $u =$ *isub-to-fsub* $\vartheta \cdot$ *isub-to-fsub* i **by** *auto*
then show $\exists i.\ u =$ *isub-to-fsub* $\vartheta \cdot i$ **by** *metis*
qed
ultimately show *?thesis* **unfolding** mgu_{ts}*-def* **by** *auto*
qed

lemma *unification′*:
assumes *finite ts*
assumes $unifier_{ts}\ \sigma\ ts$
shows $\exists \vartheta.\ mgu_{ts}\ \vartheta\ ts$
proof −
let *?E* = *fterm-to-iterm* ‘ *ts* $\times$ *fterm-to-iterm* ‘ *ts*
let *?lE* = *set-to-list ?E*
from *assms* **have** *fsub-to-isub* $\sigma \in$ *unifiers ?E* **using** *unifiert-unifiers* **by** *auto*
then have $\exists \vartheta.$ *is-imgu* ϑ *?E*
using *unify-unification*[*of fsub-to-isub* σ *?lE*] *assms* **by** (*simp add*: *set-set-to-list*)
then obtain ϑ **where** *is-imgu* ϑ *?E* **unfolding** *set-to-list-def* **by** *auto*
then have mgu_{ts} (*isub-to-fsub* ϑ) *ts* **using** *assms is-mgu-mgu*$_{ts}$ **by** *auto*
then show *?thesis* **by** *auto*
qed

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*
by (*cases l*) *auto*

lemma *literal-to-term-sub*: *literal-to-term* ($l \cdot_l \sigma$) = (*literal-to-term l*) $\cdot_t \sigma$
by (*induction l*) *auto*

lemma $unifier_{ls}$-$unifier_{ts}$:

assumes $unifier_{ls}\ \sigma\ L$
shows $unifier_{ts}\ \sigma\ (literal\text{-}to\text{-}term\ `\ L)$
proof −
from *assms* **obtain** l' **where** $\forall\, l{\in}L.\ l \cdot_l \sigma = l'$ **unfolding** $unifier_{ls}$*-def* **by** *auto*
then have $\forall\, l{\in}L.\ literal\text{-}to\text{-}term\ (l \cdot_l \sigma) = literal\text{-}to\text{-}term\ l'$ **by** *auto*
then have $\forall\, l{\in}L.\ (literal\text{-}to\text{-}term\ l) \cdot_t \sigma = literal\text{-}to\text{-}term\ l'$ **using** *literal-to-term-sub* **by** *auto*
then have $\forall\, t{\in}literal\text{-}to\text{-}term\ `\ L.\ t \cdot_t \sigma = literal\text{-}to\text{-}term\ l'$ **by** *auto*
then show *?thesis* **unfolding** $unifier_{ts}$*-def* **by** *auto*
qed

lemma $unifiert\text{-}unifier_{ls}$:
assumes $unifier_{ts}\ \sigma\ (literal\text{-}to\text{-}term\ `\ L)$
shows $unifier_{ls}\ \sigma\ L$
proof −
from *assms* **obtain** t' **where** $\forall\, t{\in}literal\text{-}to\text{-}term\ `\ L.\ t \cdot_t \sigma = t'$ **unfolding** $unifier_{ts}$*-def* **by** *auto*
then have $\forall\, t{\in}literal\text{-}to\text{-}term\ `\ L.\ term\text{-}to\text{-}literal\ (t \cdot_t \sigma) = term\text{-}to\text{-}literal\ t'$ **by** *auto*
then have $\forall\, l{\in}\ L.\ term\text{-}to\text{-}literal\ ((literal\text{-}to\text{-}term\ l) \cdot_t \sigma) = term\text{-}to\text{-}literal\ t'$ **by** *auto*
then have $\forall\, l{\in}\ L.\ term\text{-}to\text{-}literal\ ((literal\text{-}to\text{-}term\ (l \cdot_l \sigma))) = term\text{-}to\text{-}literal\ t'$ **using** *literal-to-term-sub* **by** *auto*
then have $\forall\, l{\in}\ L.\ l \cdot_l \sigma = term\text{-}to\text{-}literal\ t'$ **by** *auto*
then show *?thesis* **unfolding** $unifier_{ls}$*-def* **by** *auto*
qed

lemma mgu_{ts}-mgu_{ls}:
assumes $mgu_{ts}\ \vartheta\ (literal\text{-}to\text{-}term\ `\ L)$
shows $mgu_{ls}\ \vartheta\ L$
proof −
from *assms* **have** $unifier_{ts}\ \vartheta\ (literal\text{-}to\text{-}term\ `\ L)$ **unfolding** mgu_{ts}*-def* **by** *auto*
then have $unifier_{ls}\ \vartheta\ L$ **using** $unifiert\text{-}unifier_{ls}$ **by** *auto*
moreover
{
fix u
assume $unifier_{ls}\ u\ L$
then have $unifier_{ts}\ u\ (literal\text{-}to\text{-}term\ `\ L)$ **using** $unifier_{ls}$-$unifier_{ts}$ **by** *auto*
then have $\exists\, i.\ u = \vartheta \cdot i$ **using** *assms* **unfolding** mgu_{ts}*-def* **by** *auto*
}
ultimately show *?thesis* **unfolding** mgu_{ls}*-def* **by** *auto*
qed

theorem *unification*:
assumes *fin*: *finite L*
assumes *uni*: $unifier_{ls}\ \sigma\ L$
shows $\exists\, \vartheta.\ mgu_{ls}\ \vartheta\ L$
proof −
from *uni* **have** $unifier_{ts}\ \sigma\ (literal\text{-}to\text{-}term\ `\ L)$ **using** $unifier_{ls}$-$unifier_{ts}$ **by** *auto*

then obtain ϑ **where** $mgu_{ts}\ \vartheta$ (*literal-to-term* ‘ L) **using** *fin unification'* **by** *blast*
then have $mgu_{ls}\ \vartheta\ L$ **using** mgu_{ts}-mgu_{ls} **by** *auto*
then show *?thesis* **by** *auto*
qed

end

20 Instance of completeness theorem

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

interpretation *unification* **using** *unification* **by** *unfold-locales auto*

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* $C'\ D'\ L'\ M'\ \sigma$
shows $\exists L\ M\ \tau$. *applicable* $C\ D\ L\ M\ \tau \wedge$
instance-of$_{ls}$ (*resolution* $C'\ D'\ L'\ M'\ \sigma$) (*resolution* $C\ D\ L\ M\ \tau$)
using *assms lifting* **by** *metis*

thm *completeness*

theorem *complete*:
assumes *finite-cs*: *finite* $Cs \wedge C{\in}Cs$. *finite* C
assumes *unsat*: $\forall$ (F::*hterm fun-denot*) (G::*hterm pred-denot*) . $\neg eval_{cs}\ F\ G\ Cs$
shows $\exists Cs'$. *resolution-deriv* $Cs\ Cs' \wedge \{\} \in Cs'$
using *assms completeness* **by** −

thm *completeness-countable*

theorem *complete-countable*:
assumes *inf-uni*: *infinite* (*UNIV* :: ($'u$ :: *countable*) *set*)
assumes *finite-cs*: *finite* $Cs \wedge C{\in}Cs$. *finite* C
assumes *unsat*: $\forall$ (F::$'u$ *fun-denot*) (G::$'u$ *pred-denot*). $\neg eval_{cs}\ F\ G\ Cs$
shows $\exists Cs'$. *resolution-deriv* $Cs\ Cs' \wedge \{\} \in Cs'$
using *assms completeness-countable* **by** −

thm *completeness-nat*

theorem *complete-nat*:
assumes *finite-cs*: *finite* $Cs \wedge C{\in}Cs$. *finite* C
assumes *unsat*: $\forall$ (F::*nat fun-denot*) (G::*nat pred-denot*) . $\neg eval_{cs}\ F\ G\ Cs$

shows $\exists\, Cs'.\ \mathit{resolution\text{-}deriv}\ Cs\ Cs' \wedge \{\} \in Cs'$
using *assms completeness-nat* **by** −

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.