Technical University of Denmark

The Resolution Calculus for First-Order Logic

Schlichtkrull, Anders

Published in:
The Archive of Formal Proofs

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

June 30, 2016

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 at the International Conference on Interactive Theorem Proving [7] and an earlier version in an MSc thesis [6]. It mostly follows textbooks by Ben-Ari [1], Chang and Lee [3], and Leitsch [4]. The theory is part of the IsaFoL project [2].

Contents

1 Terms and Literals

theory *TermsAndLiterals* **imports** *Main* ~~/*src*/*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*: $ground_l$ *l* ⟹ *get-atom l* ∈ *ground-fatoms*
  unfolding *ground-fatoms-def* **by** (*induction l*) *auto*

1.2 Auxiliary

lemma *infinity*:
  assumes *inj*: ∀ *n* :: *nat*. *undiago* (*diago n*) = *n*
  assumes *all-tree*: ∀ *n* :: *nat*. (*diago n*) ∈ *S*
  shows ¬*finite S*
proof −
  from *inj all-tree* **have** ∀ *n*. *n* = *undiago* (*diago n*) ∧ (*diago n*) ∈ *S* **by** *auto*
  then have ∀ *n*. ∃ *ds*. *n* = *undiago ds* ∧ *ds* ∈ *S* **by** *auto*

then have *undiago* ' S = (*UNIV* :: *nat set*) **by** *auto*
then show ¬*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-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-into A f*) *b*) = *b*
using *assms bij-betw-inv-into-right* **by** *metis*

1.3 Conversions

1.3.1 Convertions - 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 [*simp*]: *hterm-of-fterm* (*fterm-of-hterm t*) = *t*
by (*induction t*) (*simp add*: *map-idI*)

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

lemma [*simp*]: $ground_t$ *t* ⟹ *fterm-of-hterm* (*hterm-of-fterm t*) = *t*
by (*induction t*) (*auto simp add*: *map-idI*)

lemma [*simp*]: $ground_{ts}$ *ts* ⟹ *fterms-of-hterms* (*hterms-of-fterms ts*) = *ts*
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*]: $ground_l$ *l* ⟹ *flit-of-hlit* (*hlit-of-flit l*) = *l* **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 $\forall l. \exists l'.$ $ground_l$ $l' \wedge l =$ *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 $\forall x \in UNIV. \forall y \in UNIV.$ *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 $\forall l.$ $ground_l$ *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 $\forall l.$ $ground_l$ (*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 Convertions - 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*]: $ground_{ts}$ (*snd l*) ⟹ *fatom-of-hatom* (*hatom-of-fatom l*) = *l* **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 ∀ *x*∈*UNIV*. ∀ *y*∈*UNIV*. *fatom-of-hatom x* = *fatom-of-hatom y* ⟶ *x* = *y*
using *ground-fatom-of-hatom hatom-of-fatom-fatom-of-hatom* **by** *metis*
next
have ∀ *a*. $ground_{ts}$ (*snd a*) ⟶ (*a* = *fatom-of-hatom* (*hatom-of-fatom a*)) **using** *hatom-of-fatom-fatom-of-hatom* **by** *auto*
then have *ground-fatoms* ⊆ *fatom-of-hatom* ‘ *UNIV* **unfolding** *ground-fatoms-def* **by** *blast*
moreover
have ∀ *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-from-string*:: *string* ⇒ *nat* **where**

nat-from-string ≡ (*SOME f. bij f*)

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

lemma *nat-from-string-bij*: *bij nat-from-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* ⇒ *nat*) **using** *countableE-infinite*[*of UNIV*] **by** *blast*
then show *?thesis* **unfolding** *nat-from-string-def* **using** *someI* **by** *metis*
qed

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

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

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

1.4.2 Enumerating Herbrand Atoms

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

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

instantiation *hterm* :: *countable* **begin**
instance by *countable-datatype*
end

lemma *infinite-hatoms*: *infinite* (*UNIV* :: (*pred-sym* ∗ *'t list*) *set*)
proof −
let *?diago* = *λn.* (*string-from-nat n*,[])
let *?undiago* = *λa. nat-from-string* (*fst a*)
have ∀ *n. ?undiago* (*?diago n*) = *n* **using** *nat-from-string-string-from-nat* **by** *auto*
moreover
have ∀ *n. ?diago n* ∈ *UNIV* **by** *auto*

ultimately show *infinite* (*UNIV* :: (*pred-sym* ∗ *'t list*) *set*) **using** *infinity*[*of ?undiago ?diago UNIV*] **by** *simp*
qed

lemma *nat-from-hatom-bij*: *bij nat-from-hatom*
proof −
let *?S* = *UNIV* :: (*pred-sym* ∗ (*'t*::*countable*) *list*) *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-from-hatom* **unfolding** *nat-from-hatom-def* **using** *someI* **by** *metis*
then show *?thesis* **unfolding** *bij-betw-def inj-on-def* **unfolding** *nat-from-hatom-def* **by** *simp*
qed

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

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

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

1.4.3 Enumerating Ground Atoms

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

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

theorem *diag-undiag-fatom*[*simp*]: $ground_{ts}$ *ts* ⟹ *fatom-from-nat* (*nat-from-fatom* (*p*,*ts*)) = (*p*,*ts*)
unfolding *fatom-from-nat-def nat-from-fatom-def* **by** *auto*

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

lemma *fatom-from-nat-bij*: *bij-betw fatom-from-nat UNIV ground-fatoms*
using *hatom-from-nat-bij bij-betw-trans fatom-of-hatom-bij hatom-from-nat-bij*

unfolding *fatom-from-nat-def comp-def* **by** *blast*

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

lemma *nat-from-fatom-bij*: *bij-betw nat-from-fatom ground-fatoms UNIV*
using *nat-from-hatom-bij bij-betw-trans hatom-of-fatom-bij hatom-from-nat-bij*
unfolding *nat-from-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*: *treesize T* = *0* ⟹ *T* = *Leaf* **by** (*cases T*) *auto*

lemma *treesize-Branching*: *treesize T* = *Suc n* ⟹ ∃ *l r*. *T* = *Branching l r* **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* $(a\#ds)$ $T \longrightarrow (\exists\, l\ r.\ T{=}Branching\ l\ r)$
by (*cases T*) (*auto simp add*: *path-inv-Leaf*)

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

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

lemma *path-inv-Branching*:
path p $(Branching\ l\ r) \longleftrightarrow (p{=}[] \lor (\exists\, a\ p'.\ p{=}a\#p' \land (a \longrightarrow path\ p'\ l) \land (\neg a \longrightarrow path\ p'\ r)))$ (**is** $?L \longleftrightarrow ?R$)
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 $\exists\, a\ p'.\ p{=}a\#p' \land (a \longrightarrow path\ p'\ l) \land (\neg a \longrightarrow path\ p'\ r)$
then obtain $a\ p'$ **where** $p{=}a\#p' \land (a \longrightarrow path\ p'\ l) \land (\neg a \longrightarrow path\ p'\ r)$ **by** *auto*
then show $?L$ **by** (*cases a*) *auto*
qed
qed

lemma *path-prefix*: *path* $(ds1 @ ds2)$ $T \Longrightarrow path\ ds1\ T$
proof (*induction ds1 arbitrary*: *T*)
case (*Cons a ds1*)
then have $\exists\, l\ r.\ T = Branching\ l\ r$ **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*: $\neg a$
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 T₁ T₂)
  then obtain b where branch b T₁ by auto
  then have branch (Left#b) (Branching T₁ T₂)  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 $\forall b.\ \mathit{branch}\ b\ T \longrightarrow \neg(\exists l\ r.\ \mathit{branch}\ b\ (\mathit{Branching}\ l\ r))$
using *branch-inv-Branching* **by** *auto*
then have $T=\mathit{Leaf}$ **using** *has-branch*[*of T*] **by** (*metis branch.elims*(*2*))
}
ultimately show $T = \mathit{Leaf} \longleftrightarrow (\forall b.\ \mathit{branch}\ b\ T \longrightarrow b = [])$ **by** *auto*
qed

lemma *branch-is-path*:
$\mathit{branch}\ ds\ T \Longrightarrow \mathit{path}\ ds\ T$
proof (*induction T arbitrary*: *ds*)
case *Leaf*
then have $ds = []$ **using** *branch-inv-Leaf* **by** *auto*
then show *?case* **by** *auto*
next
case $(\mathit{Branching}\ T_1\ T_2)$
then obtain *a b* **where** *ds-p*: $ds = a \# b \wedge (a \longrightarrow \mathit{branch}\ b\ T_1) \wedge (\neg\ a \longrightarrow \mathit{branch}\ b\ T_2)$ **using** *branch-inv-Branching*[*of ds*] **by** *blast*
then have $(a \longrightarrow \mathit{path}\ b\ T_1) \wedge (\neg a \longrightarrow \mathit{path}\ b\ T_2)$ **using** *Branching* **by** *auto*
then show *?case* **using** *ds-p* **by** (*cases a*) *auto*
qed

lemma *Branching-Leaf-Leaf-Tree*: $T = \mathit{Branching}\ T1\ T2 \Longrightarrow (\exists B.\ \mathit{branch}\ (B@[\mathit{True}])\ T \wedge \mathit{branch}\ (B@[\mathit{False}])\ T)$
proof (*induction T arbitrary*: *T1 T2*)
case *Leaf* **then show** *?case* **by** *auto*
next
case (*Branching T1' T2'*)
{
assume $T1'=\mathit{Leaf} \wedge T2'=\mathit{Leaf}$
then have $\mathit{branch}\ ([]\ @\ [\mathit{True}])\ (\mathit{Branching}\ T1'\ T2') \wedge \mathit{branch}\ ([]\ @\ [\mathit{False}])\ (\mathit{Branching}\ T1'\ T2')$ **by** *auto*
then have *?case* **by** *metis*
}
moreover
{
fix *T11 T12*
assume $T1' = \mathit{Branching}\ T11\ T12$
then obtain *B* **where** $\mathit{branch}\ (B\ @\ [\mathit{True}])\ T1'$
$\wedge\ \mathit{branch}\ (B\ @\ [\mathit{False}])\ T1'$ **using** *Branching* **by** *blast*
then have $\mathit{branch}\ (([\mathit{True}]\ @\ B)\ @\ [\mathit{True}])\ (\mathit{Branching}\ T1'\ T2')$
$\wedge\ \mathit{branch}\ (([\mathit{True}]\ @\ B)\ @\ [\mathit{False}])\ (\mathit{Branching}\ T1'\ T2')$ **by** *auto*
then have *?case* **by** *blast*
}
moreover
{
fix *T11 T12*
assume $T2' = \mathit{Branching}\ T11\ T12$
then obtain *B* **where** $\mathit{branch}\ (B\ @\ [\mathit{True}])\ T2'$
$\wedge\ \mathit{branch}\ (B\ @\ [\mathit{False}])\ T2'$ **using** *Branching* **by** *blast*

then have *branch* $(([False] @ B) @ [True])$ $(Branching\ T1'\ T2')$
$\wedge$ *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* $\Rightarrow$ *tree* $\Rightarrow$ *bool* **where**
internal [] $(Branching\ l\ r) \longleftrightarrow True$
| *internal* $(d\#ds)$ $(Branching\ l\ r) \longleftrightarrow (if\ d\ then\ internal\ ds\ l\ else\ internal\ ds\ r)$
| *internal* - - $\longleftrightarrow$ *False*

lemma *internal-inv-Leaf*: $\neg$*internal b Leaf* **using** *internal.simps* **by** *blast*

lemma *internal-inv-Branching-Left*:
internal $(Left\#b)$ $(Branching\ l\ r) \longleftrightarrow internal\ b\ l$ **by** *auto*

lemma *internal-inv-Branching-Right*:
internal $(Right\#b)$ $(Branching\ l\ r) \longleftrightarrow internal\ b\ r$
by *auto*

lemma *internal-inv-Branching*:
internal p $(Branching\ l\ r) \longleftrightarrow (p=[] \vee (\exists\, a\ p'.\ p=a\#p' \wedge (a \longrightarrow internal\ p'\ l)$
$\wedge (\neg a \longrightarrow internal\ p'\ r)))$ (**is** *?L* $\longleftrightarrow$ *?R*)
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 $\exists\, a\ p'.\ p=a\#p' \wedge (a \longrightarrow internal\ p'\ l) \wedge (\neg a \longrightarrow internal\ p'\ r)$
then obtain $a\ p'$ **where** $p=a\#p' \wedge (a \longrightarrow internal\ p'\ l) \wedge (\neg a \longrightarrow internal$
$p'\ r)$ **by** *auto*
then show *?L* **by** (*cases a*) *auto*
qed
qed

lemma *internal-is-path*:
internal ds T $\Longrightarrow$ *path ds T*
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*: $ds=[] \vee ds = a \# b \wedge (a \longrightarrow internal\ b\ T_1) \wedge (\neg\ a \longrightarrow internal\ b\ T_2)$ **using** *internal-inv-Branching* **by** *blast*
then have $ds = [] \vee (a \longrightarrow path\ b\ T_1) \wedge (\neg a \longrightarrow path\ b\ T_2)$ **using** *Branching* **by** *auto*
then show *?case* **using** *ds-p* **by** (*cases a*) *auto*
qed

lemma *internal-prefix*: *internal* (*ds1*@*ds2*@[*d*]) $T \Longrightarrow$ *internal ds1 T*
proof (*induction ds1 arbitrary*: *T*)
case (*Cons a ds1*)
then have $\exists\, l\ r.\ T = Branching\ l\ r$ **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 $\exists\, l\ r.\ T = Branching\ l\ r$ **using** *internal-inv-Leaf* **by** (*cases T*) *auto*
then show *?case* **by** *auto*
qed

lemma *internal-branch*: *branch* (*ds1*@*ds2*@[*d*]) $T \Longrightarrow$ *internal ds1 T*
proof (*induction ds1 arbitrary*: *T*)
case (*Cons a ds1*)
then have $\exists\, l\ r.\ T = Branching\ l\ r$ **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 $\exists$ *l r*. *T* = *Branching l r* **using** *branch-inv-Leaf* **by** (*cases T*) *auto*
then show *?case* **by** *auto*
qed

fun *parent* :: *dir list* $\Rightarrow$ *dir list* **where**
parent ds = *tl ds*

2.5 Deleting Nodes

fun *delete* :: *dir list* $\Rightarrow$ *tree* $\Rightarrow$ *tree* **where**
delete [] *T* = *Leaf*
| *delete* (*True*#*ds*) (*Branching* T_1 T_2) = *Branching* (*delete ds* T_1) T_2
| *delete* (*False*#*ds*) (*Branching* T_1 T_2) = *Branching* T_1 (*delete ds* T_2)
| *delete* (*a*#*ds*) *Leaf* = *Leaf*

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

lemma *path-delete*: *path p* (*delete ds T*) $\Longrightarrow$ *path p T*
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 $\exists$ *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 $\exists$ *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*: $\neg$*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*: $\neg a$
  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*: $\neg a$
  assume *b-p*: $\neg b$
  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*: *branch p* (*delete ds T*) $\Longrightarrow$ *branch p T* $\vee$ *p=ds*
proof (*induction p arbitrary*: *T ds*)
  case *Nil*
  then have *delete ds T* = *Leaf* **by** (*cases delete ds T*) *auto*
  then have *ds* = [] $\vee$ *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 $\exists$ *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 $\exists$ *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*: *path p* (*delete ds T*) ⟹ ¬(∃ *c cs*. *p* = *ds* @ *c*#*cs*)

```
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*: *internal p T* $\Longrightarrow$ *treesize* (*delete p T*) < *treesize T*
proof (*induction p arbitrary*: *T*)
  case (*Nil*)
  then have $\exists$ *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 $\exists$ *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* $\Rightarrow$ *bool*) $\Rightarrow$ *dir list* $\Rightarrow$ *tree* $\Rightarrow$ *tree* **where**
  cutoff red ds (*Branching* T_1 T_2) =
    (*if red ds then Leaf else Branching* (*cutoff red* (*ds*@[*Left*]) T_1) (*cutoff red* (*ds*@[*Right*]) T_2))
| *cutoff red ds Leaf* = *Leaf*

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

lemma *treesize-cutoff*: *treesize* (*cutoff red ds T*) $\leq$ *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*) $\leq$ *treesize T1* + *treesize T2* **using** *add-mono* **by** *blast*
then show *?case* **by** *auto*
qed

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

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

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

lemma *cutoff-branch′*:
anybranch T (λ*b*. *red*(*ds*@*b*)) $\Longrightarrow$ *anybranch* (*cutoff red ds T*) (λ*b*. *red*(*ds*@*b*))
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** $\forall$ *p*. *branch* (*Left*#*p*) (*Branching* T_1 T_2) $\longrightarrow$ *red* (*ds* @ (*Left*#*p*)) **by** *blast*
then have $\forall$ *p*. *branch p* T_1 $\longrightarrow$ *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** $\forall$ *p*. *branch* (*Right*#*p*) (*Branching* T_1 T_2) $\longrightarrow$ *red* (*ds* @ (*Right*#*p*)) **by** *blast*
then have $\forall$ *p*. *branch p* T_2 $\longrightarrow$ *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* $\vee$ $\neg$*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* $?T_1{}'$ $?T_2{}'$ **by** *auto*
from *this b-p* **obtain** *a* b' **where** $b = a \# b' \wedge (a \longrightarrow$ *branch* b' $?T_1{}') \wedge (\neg a \longrightarrow$ *branch* b' $?T_2{}'$) **using** *branch-inv-Branching*[*of b* $?T_1{}'$ $?T_2{}'$] **by** *auto*
then show *red*(*ds*@*b*) **using** *aa bb* **by** (*cases a*) *auto*
qed
}
then show *?case* **by** *blast*
qed

lemma *cutoff-branch*: *anybranch T* (λp. *red p*) $\Longrightarrow$ *anybranch* (*cutoff red* [] *T*) (λp. *red p*)
using *cutoff-branch'*[*of T red* []] **by** *auto*

lemma *cutoff-internal'*:
anybranch T (λb. *red*(*ds*@*b*)) $\Longrightarrow$ *anyinternal* (*cutoff red ds T*) (λb. ¬*red*(*ds*@*b*))
proof (*induction T arbitrary*: *ds*)
case (*Leaf*) **then show** *?case* **using** *internal-inv-Leaf* **by** *simp*
next
case (*Branching* T_1 T_2)
let *?T* = *cutoff red ds* (*Branching* T_1 T_2)
from *Branching* **have** $\forall p$. *branch* (*Left*#*p*) (*Branching* T_1 T_2) $\longrightarrow$ *red* (*ds* @ (*Left*#*p*)) **by** *blast*
then have $\forall p$. *branch p* T_1 $\longrightarrow$ *red* (*ds* @ (*Left*#*p*)) **by** *auto*
then have *anybranch* T_1 (λp. *red* ((*ds* @ [*Left*]) @ *p*)) **by** *auto*
then have *aa*: *anyinternal* (*cutoff red* (*ds* @ [*Left*]) T_1) (λp. ¬ *red* ((*ds* @ [*Left*]) @ *p*)) **using** *Branching* **by** *blast*

from *Branching* **have** $\forall p$. *branch* (*Right*#*p*) (*Branching* T_1 T_2) $\longrightarrow$ *red* (*ds* @ (*Right*#*p*)) **by** *blast*
then have $\forall p$. *branch p* T_2 $\longrightarrow$ *red* (*ds* @ (*Right*#*p*)) **by** *auto*
then have *anybranch* T_2 (λp. *red* ((*ds* @ [*Right*]) @ *p*)) **by** *auto*
then have *bb*: *anyinternal* (*cutoff red* (*ds* @ [*Right*]) T_2) (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 $p=[] \vee p\neq[]$ **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*]) T_1
let *?T₂'* = *cutoff red* (*ds*@[*Right*]) T_2
assume $p \neq []$
moreover
have *?T* = *Branching ?T₁' ?T₂'* **using** *ds-p* **by** *auto*
ultimately
obtain *a p'* **where** *b-p*: *p* = *a* # *p'* ∧
(*a* ⟶ *internal p'* (*cutoff red* (*ds* @ [*Left*]) T_1)) ∧
(¬ *a* ⟶ *internal p'* (*cutoff red* (*ds* @ [*Right*]) T_2))
using *b-p internal-inv-Branching*[*of p ?T₁' ?T₂'*] **by** *auto*
then have ¬*red*(*ds* @ [*a*] @ *p'*) **using** *aa bb* **by** (*cases a*) *auto*
then show ¬*red*(*ds* @ *p*) **using** *b-p* **by** *simp*
qed
}
then show *?case* **by** *blast*
qed

lemma *cutoff-internal*: *anybranch T red* ⟹ *anyinternal* (*cutoff red* [] *T*) (λ*p*. ¬*red p*)
using *cutoff-internal'*[*of T red* []] **by** *auto*

lemma *cutoff-branch-internal'*:
anybranch T red ⟹ *anyinternal* (*cutoff red* [] *T*) (λ*p*. ¬*red p*) ∧ *anybranch* (*cutoff red* [] *T*) (λ*p*. *red p*)
using *cutoff-internal*[*of T*] *cutoff-branch*[*of T*] **by** *blast*

lemma *cutoff-branch-internal*:
anybranch T red ⟹ ∃ *T'*. *anyinternal T'* (λ*p*. ¬*red p*) ∧ *anybranch T'* (λ*p*. *red p*)
using *cutoff-branch-internal'* **by** *blast*

3 Possibly Infinite Trees

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

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

The subtree in with root r

fun *subtree* :: *dir list set* ⇒ *dir list* ⇒ *dir list set* **where**
subtree T r = {*ds* ∈ *T*. ∃ *ds'*. *ds* = *r* @ *ds'*}

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

lemma *subtree-pos*:
subtree T ds ⊆ *subtree T* (*ds* @ [*Left*]) ∪ *subtree T* (*ds* @ [*Right*]) ∪ {*ds*}

proof (*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 $(\exists\, e.\ x = ds\ @\ [Left]\ @\ e) \lor (\exists\, e.\ x = ds\ @\ [Right]\ @\ e)$ **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 $\equiv (f\ 0 = []) \land (\forall\, n.\ \exists\, a.\ f\ (Suc\ n) = (f\ n)\ @\ [a])$

lemma *infpath-length*: *wf-infpath f* $\Longrightarrow$ *length* $(f\ n) = n$
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*: *wf-infpath f* $\Longrightarrow n_1 \leq n_2 \Longrightarrow \exists\, a.\ (f\ n_1)\ @\ a = (f\ n_2)$
proof (*induction* n_2)
  case (*Suc* n_2)
  then have $n_1 \leq n_2 \lor n_1 = Suc\ n_2$ **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 $\exists\, c.$ *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*: $\bigwedge$*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[of []] by auto
      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[of ?nextnode (?c n) suf] by auto

      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: (l^c)^c = l by (cases l) auto

lemma cancel-comp2:
  assumes asm: l1^c = l2^c
  shows l1 = l2
proof -
  from asm have (l1^c)^c = (l2^c)^c by auto
  then have l1 = (l2^c)^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* $\Rightarrow$ *var-sym set* **where**
$vars_{ts}\ ts \equiv (\bigcup t \in set\ ts.\ vars_t\ t)$

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

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

lemma *ground-vars*$_t$: $ground_t\ t \Longrightarrow vars_t\ t = \{\}$
by (*induction t*) *auto*

lemma $ground_{ts}$-$vars_{ts}$: $ground_{ts}\ ts \Longrightarrow vars_{ts}\ ts = \{\}$
using *ground-vars*$_t$ **by** *auto*

lemma $ground_l$-$vars_l$: $ground_l\ l \Longrightarrow vars_l\ l = \{\}$ **unfolding** $vars_l$-*def* **using** *ground-vars*$_t$ **by** *auto*

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

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

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

7 Semantics

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

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

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

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

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

definition $eval_{cs}$ :: *'u fun-denot* $\Rightarrow$ *'u pred-denot* $\Rightarrow$ *fterm clause set* $\Rightarrow$ *bool* **where**

$eval_{cs}\ F\ G\ Cs \longleftrightarrow (\forall\, C \in Cs.\ eval_c\ F\ G\ C)$

7.1 Semantics of Ground Terms

lemma *ground-var-denott*: $ground_t\ t \Longrightarrow (eval_t\ E\ F\ t = eval_t\ E'\ F\ t)$
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*: $ground_{ts}\ ts \Longrightarrow (eval_{ts}\ E\ F\ ts = eval_{ts}\ E'\ F\ ts)$
  using *ground-var-denott* **by** (*metis map-eq-conv*)

lemma *ground-var-denot*: $ground_l\ l \Longrightarrow (eval_l\ E\ F\ G\ l = eval_l\ E'\ F\ G\ l)$
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** $\cdot_t$ *55*) **where**
  $(Var\ x) \cdot_t \sigma = \sigma\ x$
| $(Fun\ f\ ts) \cdot_t \sigma = Fun\ f\ (map\ (\lambda t.\ t \cdot_t \sigma)\ ts)$

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

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

abbreviation *subls* :: *fterm literal set* ⇒ *substitution* ⇒ *fterm literal set* (**infixl** $\cdot_{ls}$ *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* $\Rightarrow$ *fterm* $\Rightarrow$ *bool* **where**
instance-of$_t$ t_1 t_2 $\longleftrightarrow$ $(\exists \sigma.\ t_1 = t_2 \cdot_t \sigma)$

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

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

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

lemma *comp-sub*: $(l^c) \cdot_l \sigma = (l \cdot_l \sigma)^c$
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* $\Rightarrow$ *fterm clause* $\Rightarrow$ *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*: *ground*$_t$ $t \Longrightarrow t \cdot_t \sigma = t$
by (*induction t*) (*auto simp add*: *map-idI*)

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

lemma *ground*$_l$*-subs*: *ground*$_l$ $l \Longrightarrow l \cdot_l \sigma = l$
using *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* $\Rightarrow$ *substitution* $\Rightarrow$ *substitution* (**infixl** $\cdot$ *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 $((\mathit{Var}\ x) \cdot_t \sigma_1) \cdot_t \sigma_2 = (\sigma_1\ x) \cdot_t \sigma_2$ **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 $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3))\ x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3)\ x$ **unfolding** *composition-def* **using** *composition-conseq2t* **by** *simp*
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 $?L \subseteq C$ **by** *auto*
moreover
have $?L \cdot_{ls} lmbd = L'$
proof (*rule Orderings.order-antisym*; *rule Set.subsetI*)
fix l'
assume $l'L$: $l' \in L'$
from *inst-C* **have** $\{l \cdot_l lmbd | l.\ l \in C\} = C'$ **unfolding** *subls-def2* **by** −
then have $\exists l.\ l' = l \cdot_l lmbd \wedge l \in C \wedge l \cdot_l lmbd \in L'$ **using** $L'sub$ $l'L$ **by** *auto*
then have $l' \in \{l \in C.\ l \cdot_l lmbd \in L'\} \cdot_{ls} lmbd$ **by** *auto*
then show $l' \in \{l \in C.\ \exists l' \in L'.\ l \cdot_l lmbd = l'\} \cdot_{ls} lmbd$ **by** *auto*
qed *auto*
moreover
have $(C - ?L) \cdot_{ls} lmbd = C' - L'$ **using** *inst-C* **by** *auto*
moreover
ultimately show *?thesis* **by** *auto*
qed

lemma *relevant-vars-subt*:

$\forall x \in vars_t\ t.\ \sigma_1\ x = \sigma_2\ x \Longrightarrow t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
proof (*induction t*)
case (*Fun f ts*)
have *f*: $\bigwedge t.\ t \in set\ ts \Longrightarrow vars_t\ t \subseteq vars_{ts}\ ts$ **by** (*induction ts*) *auto*
have $\forall t{\in}set\ ts.\ t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
proof
fix *t*
assume *tints*: $t \in set\ ts$
then have $\forall x \in vars_t\ t.\ \sigma_1\ x = \sigma_2\ x$ **using** *f Fun(2)* **by** *auto*
then show $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$ **using** *Fun tints* **by** *auto*
qed
then have $ts \cdot_{ts} \sigma_1 = ts \cdot_{ts} \sigma_2$ **by** *auto*
then show *?case* **by** *auto*
qed *auto*

lemma *relevant-vars-subts*:
assumes *asm*: $\forall x \in vars_{ts}\ ts.\ \sigma_1\ x = \sigma_2\ x$
shows $ts \cdot_{ts} \sigma_1 = ts \cdot_{ts} \sigma_2$
proof –
have *f*: $\bigwedge t.\ t \in set\ ts \Longrightarrow vars_t\ t \subseteq vars_{ts}\ ts$ **by** (*induction ts*) *auto*
have $\forall t{\in}set\ ts.\ t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
proof
fix *t*
assume *tints*: $t \in set\ ts$
then have $\forall x \in vars_t\ t.\ \sigma_1\ x = \sigma_2\ x$ **using** *f asm* **by** *auto*
then show $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$ **using** *relevant-vars-subt tints* **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *relevant-vars-subl*:
$\forall x \in vars_l\ l.\ \sigma_1\ x = \sigma_2\ x \Longrightarrow l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
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*: $\bigwedge l.\ l \in L \Longrightarrow vars_l\ l \subseteq vars_{ls}\ L$ **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* ⇒ *fterm clause* **where**
$std_2\ C \equiv C \cdot_{ls}\ (\lambda x.\ Var\ (''2''\ @\ x))$

lemma *std-apart-apart''*:
$x \in vars_t\ \ (t \cdot_t (\lambda x{::}char\ list.\ Var\ (y\ @\ x))) \Longrightarrow \exists x'.\ x = y@x'$
by (*induction t*) *auto*

lemma *std-apart-apart'*: $x \in vars_l\ (l \cdot_l (\lambda x.\ Var\ \ (y@x))) \Longrightarrow \exists x'.\ x = y@x'$
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-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** *composition-conseq2ls* **by** *auto*
then have $C_1 = (std_1\ C_1) \cdot_{ls} (\lambda x.\ Var\ (tl\ x))$ **by** *auto*
then show *instance-of*$_{ls}$ C_1 (std_1 C_1) **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
qed

lemma *std-apart-instance-of*$_{ls}$*2*: *instance-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-of*$_{ls}$ *C2* (std_2 *C2*) **unfolding** *instance-of*$_{ls}$*-def* **by** *auto*
qed

9 Unifiers

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

definition *unifier*$_{ls}$ :: *substitution* $\Rightarrow$ *fterm literal set* $\Rightarrow$ *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*$_{ls}$*-def* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *unifiert-def2*:
assumes *L-elem*: $ts \neq \{\}$
shows *unifier*$_{ts}$ σ *ts* $\longleftrightarrow$ $(\exists\, l.\ (\lambda t.\ sub\ t\ \sigma)$ ‘ $ts = \{l\})$
proof
assume *unif*: *unifier*$_{ts}$ σ *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*$_{ts}$*-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*$_{ts}$ σ *ts* **unfolding** *unifier*$_{ts}$*-def* **by** *auto*
qed

lemma *unifier*$_{ls}$*-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* $\Rightarrow$ *bool* **where**

unifiablets fs $\longleftrightarrow$ ($\exists\,\sigma$. *unifier*$_{ts}$ σ *fs*)

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

lemma *unifier-comp*[*simp*]: *unifier*$_{ls}$ σ (L^C) $\longleftrightarrow$ *unifier*$_{ls}$ σ *L*
proof
assume *unifier*$_{ls}$ σ (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*$_{ls}$ σ *L* **unfolding** *unifier*$_{ls}$-*def* **by** *auto*
next
assume *unifier*$_{ls}$ σ *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*$_{ls}$ σ (L^C) **unfolding** *unifier*$_{ls}$-*def* **by** *auto*
qed

lemma *unifier-sub1*: *unifier*$_{ls}$ σ $L \Longrightarrow L' \subseteq L \Longrightarrow$ *unifier*$_{ls}$ σ L'
unfolding *unifier*$_{ls}$-*def* **by** *auto*

lemma *unifier-sub2*:
assumes *asm*: *unifier*$_{ls}$ σ $(L_1 \cup L_2)$
shows *unifier*$_{ls}$ σ L_1 $\wedge$ *unifier*$_{ls}$ σ 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* $\Rightarrow$ *fterm set* $\Rightarrow$ *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* $\Rightarrow$ *fterm literal set* $\Rightarrow$ *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* $\Rightarrow$ *fterm clause*
$\Rightarrow$ *fterm literal set* $\Rightarrow$ *fterm literal set*
$\Rightarrow$ *substitution* $\Rightarrow$ *bool* **where**
applicable $C_1\ C_2\ L_1\ L_2\ \sigma \longleftrightarrow$
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
$\wedge\ vars_{ls}\ C_1 \cap vars_{ls}\ C_2 = \{\}$
$\wedge\ L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
$\wedge\ mgu_{ls}\ \sigma\ (L_1 \cup L_2{}^C)$

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

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

inductive *mresolution-step* :: *fterm clause set* $\Rightarrow$ *fterm clause set* $\Rightarrow$ *bool* **where**
mresolution-rule:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *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 $eval_c\ F\ G\ (C \cdot_{ls} \sigma)$
proof −
 have $\forall E.\ \exists l \in C \cdot_{ls} \sigma.\ eval_l\ E\ F\ G\ l$
  proof
   fix E
   from *asm* **have** $\forall E.\ \exists l \in C.\ eval_l\ E\ F\ G\ l$ **unfolding** $eval_c$*-def* **by** *auto*
   then have $\exists l \in C.\ eval_l\ (evalsub\ E\ F\ \sigma)\ F\ G\ l$ **by** *auto*
   then show $\exists l \in C \cdot_{ls} \sigma.\ eval_l\ E\ F\ G\ l$ **using** *substitution* **by** *blast*
  qed
 then show $eval_c\ F\ G\ (C \cdot_{ls} \sigma)$ **unfolding** $eval_c$*-def* **by** *auto*
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: $l_1 \in C_1$
  assumes l_2inc_2: $l_2 \in C_2$
  assumes *comp*: ${l_1}^c = l_2$
  shows $eval_c\ F\ G\ ((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))$
proof −
  have $\forall E.\ \exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))).\ eval_l\ E\ F\ G\ l$
    proof
      fix E
      have $eval_l\ E\ F\ G\ l_1 \lor eval_l\ E\ F\ G\ l_2$ **using** *comp* **by** (*cases* l_1) *auto*
      then show $\exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))).\ eval_l\ E\ F\ G\ l$
        proof

assume $eval_l\ E\ F\ G\ l_1$
then have $\neg eval_l\ E\ F\ G\ l_2$ **using** *comp* **by** (*cases* l_1) *auto*
then have $\exists\, l_2{}' \in C_2.\ l_2{}' \neq l_2 \wedge eval_l\ E\ F\ G\ l_2{}'$ **using** l_2inc_2 C_2sat **unfolding** $eval_c$*-def* **by** *auto*
then show $\exists\, l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}).\ eval_l\ E\ F\ G\ l$ **by** *auto*
next
assume $eval_l\ E\ F\ G\ l_2$
then have $\neg eval_l\ E\ F\ G\ l_1$ **using** *comp* **by** (*cases* l_1) *auto*
then have $\exists\, l_1{}' \in C_1.\ l_1{}' \neq l_1 \wedge eval_l\ E\ F\ G\ l_1{}'$ **using** l_1inc_1 C_1sat **unfolding** $eval_c$*-def* **by** *auto*
then show $\exists\, l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}).\ eval_l\ E\ F\ G\ l$ **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* $C_1\ C_2\ L_1\ L_2\ \sigma$
shows $eval_c\ F\ G$ (*mresolution* $C_1\ C_2\ L_1\ L_2\ \sigma$)
proof –
from sat_1 **have** $sat_1\sigma$: $eval_c\ F\ G\ (C_1 \cdot_{ls} \sigma)$ **using** *subst-sound* **by** *blast*
from sat_2 **have** $sat_2\sigma$: $eval_c\ F\ G\ (C_2 \cdot_{ls} \sigma)$ **using** *subst-sound* **by** *blast*

from *appl* **obtain** l_1 **where** l_1*-p*: $l_1 \in L_1$ **unfolding** *applicable-def* **by** *auto*

from l_1*-p* *appl* **have** $l_1 \in C_1$ **unfolding** *applicable-def* **by** *auto*
then have $inc_1\sigma$: $l_1 \cdot_l \sigma \in C_1 \cdot_{ls} \sigma$ **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** $unifier_{ls}\ \sigma\ (L_2{}^C)$
using *unifier-sub2* **unfolding** mgu_{ls}*-def* *applicable-def* **by** *blast*
then have $unifier_{ls}\ \sigma\ L_2$ **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

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

lemma *sound-step*: *mresolution-step* *Cs* Cs' $\Longrightarrow$ $eval_{cs}$ F G *Cs* $\Longrightarrow$ $eval_{cs}$ F G Cs'
proof (*induction rule*: *mresolution-step.induct*)
case (*mresolution-rule* C_1 *Cs* C_2 l_1 l_2 σ)
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 σ)
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-of*$_{ls}$*-def* **by** *metis*
  then show *?case* **using** *standardize-apart* **unfolding** $eval_{cs}$*-def* **by** *auto*
qed

lemma *lsound-step*: *resolution-step Cs Cs'* $\Longrightarrow$ $eval_{cs}$ *F G Cs* $\Longrightarrow$ $eval_{cs}$ *F G Cs'*
proof (*induction rule*: *resolution-step.induct*)
  case (*resolution-rule* C_1 *Cs* C_2 l_1 l_2 σ)
  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 σ)
    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-of*$_{ls}$*-def* **by** *metis*
  then show *?case* **using** *standardize-apart* **unfolding** $eval_{cs}$*-def* **by** *auto*
qed

lemma *sound-derivation*:
  mresolution-deriv Cs Cs' $\Longrightarrow$ $eval_{cs}$ *F G Cs* $\Longrightarrow$ $eval_{cs}$ *F G Cs'*
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** *sound-step* **by** *auto*
qed

lemma *lsound-derivation*:
  resolution-deriv Cs Cs' $\Longrightarrow$ $eval_{cs}$ *F G Cs* $\Longrightarrow$ $eval_{cs}$ *F G Cs'*
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** *lsound-step* **by** *auto*
qed

12 Herbrand Interpretations

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

term *HFun*

lemma *eval-ground$_t$*: *ground$_t$ t* $\Longrightarrow$ (*eval$_t$ E HFun t*) = *hterm-of-fterm t*
  by (*induction t*) *auto*

lemma *eval-ground$_{ts}$*: *ground$_{ts}$ ts* $\Longrightarrow$ (*eval$_{ts}$ E HFun ts*) = *hterms-of-fterms ts*
  unfolding *hterms-of-fterms-def* **using** *eval-ground$_t$* **by** (*induction ts*) *auto*

lemma *eval$_l$-ground$_{ts}$*:
  assumes *asm*: *ground$_{ts}$ ts*
  shows *eval$_l$ E HFun G* (*Pos P ts*) $\longleftrightarrow$ *G P* (*hterms-of-fterms ts*)
proof −
  have *eval$_l$ E HFun G* (*Pos P ts*) = *G P* (*eval$_{ts}$ E HFun ts*) **by** *auto*
  also have ... = *G P* (*hterms-of-fterms ts*) **using** *asm eval-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* $\Rightarrow$ *fterm literal* $\Rightarrow$ *bool* **where**
  falsifies$_l$ G l $\longleftrightarrow$
        ground$_l$ l
    $\wedge$ (*let i* = *nat-from-fatom* (*get-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* $\Rightarrow$ *fterm clause* $\Rightarrow$ *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* $\Rightarrow$ *fterm clause* $\Rightarrow$ *bool* **where**
  falsifies$_c$ G C $\equiv$ ($\exists$ *C'*. *instance-of$_{ls}$ C' C* $\wedge$ *falsifies$_g$ G C'*)

abbreviation *falsifies$_{cs}$* :: *partial-pred-denot* $\Rightarrow$ *fterm clause set* $\Rightarrow$ *bool* **where**
  falsifies$_{cs}$ G Cs $\equiv$ ($\exists$ *C* $\in$ *Cs*. *falsifies$_c$ G C*)

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

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

lemma *ground-sub-of-denott*: *ground$_t$* (*t* $\cdot_t$ (*sub-of-denot E*))

by (*induction t*) (*auto simp add*: *ground-fterm-of-hterm*)

lemma *ground-sub-of-denotts*: $ground_{ts}$ (*ts* $\cdot_{ts}$ *sub-of-denot E*)
using *ground-sub-of-denott* **by** *simp*

lemma *ground-sub-of-denotl*: $ground_l$ (*l* $\cdot_l$ *sub-of-denot E*)
proof −
  have $ground_{ts}$ (*subs* (*get-terms l*) (*sub-of-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-of-denot E x*) = *E x*
proof −
  have $ground_t$ (*sub-of-denot E x*) **using** *ground-fterm-of-hterm* **by** *simp*
  then
  have $eval_t$ *E HFun* (*sub-of-denot E x*) = *hterm-of-fterm* (*sub-of-denot E x*)
    using *eval-ground*$_t$(*1*) **by** *auto*
  also have ... = *hterm-of-fterm* (*fterm-of-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-of-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-of-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-of-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-of-denot E*) $\longleftrightarrow$ *G p* ($eval_{ts}$ *E HFun* (*ts* $\cdot_{ts}$ (*sub-of-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-of-denot E*) $\longleftrightarrow$ $\neg$*G p* ($eval_{ts}$ *E HFun* (*ts* $\cdot_{ts}$ (*sub-of-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* $\cdot_l$ *sub-of-denot E*) $\wedge$ $ground_l$ (*l* $\cdot_l$ *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* $\cdot_{ls}$ τ)
shows $falsifies_c$ *G C*
proof −
from *assms* **obtain** *C'* **where** *C'-p*: $instance\text{-}of_{ls}$ *C'* (*C* $\cdot_{ls}$ τ) $\wedge$ $falsifies_g$ *G C'* **by** *auto*
then have $instance\text{-}of_{ls}$ (*C* $\cdot_{ls}$ τ) *C* **unfolding** $instance\text{-}of_{ls}$*-def* **by** *auto*
then have $instance\text{-}of_{ls}$ *C' C* **using** *C'-p* $instance\text{-}of_{ls}$*-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*:
(($\exists$ *l* $\in$ *C*. $eval_l$ *E HFun G l*) $\longleftrightarrow$ ($\exists$ *l* $\in$ *C* $\cdot_{ls}$ *sub-of-denot E*. $eval_l$ *E HFun G l*))
$\wedge$ $ground_{ls}$ (*C* $\cdot_{ls}$ *sub-of-denot E*)
using *sub-of-denot-equiv-ground'* **by** *auto*

lemma std_1*-falsifies*: $falsifies_c$ *G* C_1 $\longleftrightarrow$ $falsifies_c$ *G* (std_1 C_1)
proof
assume *asm*: $falsifies_c$ *G* C_1
then obtain *Cg* **where** $instance\text{-}of_{ls}$ *Cg* C_1 $\wedge$ $falsifies_g$ *G Cg* **by** *auto*
moreover
then have $instance\text{-}of_{ls}$ *Cg* (std_1 C_1) **using** *std-apart-instance-of*$_{ls}$*1* $instance\text{-}of_{ls}$*-trans asm* **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*: $instance\text{-}of_{ls}$ (std_1 C_1) C_1 **unfolding** $instance\text{-}of_{ls}$*-def* **by** *auto*

from *asm* **obtain** *Cg* **where** *instance-of*$_{ls}$ *Cg* (std_1 C_1) $\wedge$ *falsifies*$_g$ *G Cg* **by** *auto*
moreover
then have *instance-of*$_{ls}$ *Cg* C_1 **using** *inst instance-of*$_{ls}$*-trans assms* **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 asm* **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 assms* **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 assms* **by** *auto*
moreover have *instance-of*$_{ls}$ (std_1 C_1) C_1 **using** *assms* **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 assms* **by** *auto*
moreover have *instance-of*$_{ls}$ (std_2 C_2) C_2 **using** *assms* **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* ∧ $\textit{falsifies}_{cs}$ *G Cs*

abbreviation(*input*) *open-branch* :: *partial-pred-denot* ⇒ *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
open-branch G T Cs ≡ *branch G T* ∧ ¬$\textit{falsifies}_{cs}$ *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*. ¬$\textit{falsifies}_{cs}$ *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*: $\textit{ground}_{ls}$ *C*
assumes *C-sat*: ¬$\textit{falsifies}_{c}$ (*f* (*Suc n*)) *C*
assumes *n-max*: ∀ *l*∈*C*. *nat-from-fatom* (*get-atom l*) ≤ *n*
shows $\textit{eval}_{c}$ *HFun* (*extend f*) *C*
proof −
let *?F* = *HFun*
let *?G* = *extend f*
{
fix *E*
from *C-sat* **have** ∀ *C′*. (¬$\textit{instance-of}_{ls}$ *C′ C* ∨ ¬$\textit{falsifies}_{g}$ (*f* (*Suc n*)) *C′*) **by** *auto*
then have ¬$\textit{falsifies}_{g}$ (*f* (*Suc n*)) *C* **using** $\textit{instance-of}_{ls}$*-self* **by** *auto*
then obtain *l* **where** *l-p*: *l*∈*C* ∧ ¬$\textit{falsifies}_{l}$ (*f* (*Suc n*)) *l* **using** *C-ground* **by** *blast*
let *?i* = *nat-from-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 $\textit{eval}_{l}$ *E HFun* (*extend f*) *l*
proof (*cases l*)
case (*Pos P ts*)
from *Pos l-p C-ground* **have** *ts-ground*: $\textit{ground}_{ts}$ *ts* **by** *auto*

have $\neg$*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-from-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 $\neg$*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 $\neg$*?G P* (*hterms-of-fterms ts*) **using** *Neg* **by** (*simp add*: *nat-from-fatom-def*)

then show *?thesis* **using** *Neg eval*$_l$*-ground*$_{ts}$[*of ts - ?G P*] *ts-ground* **by**
auto
qed
then have $\exists$ *l* $\in$ *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* $\Rightarrow$ *partial-pred-denot*)
assumes *C-ground*: *ground*$_{ls}$ *C*
assumes *fin-c*: *finite C*
assumes *model-C*: $\forall$ *n*. $\neg$*falsifies*$_c$ (*f n*) *C*
shows *C-false*: *eval*$_c$ *HFun* (*extend f*) *C*
proof −
— Since C is finite, C has a largest index of a literal.
obtain *n* **where** *largest*: $\forall$ *l* $\in$ *C*. *nat-from-fatom* (*get-atom l*) $\leq$ *n* **using** *fin-c*
maximum[*of C* λ*l*. *nat-from-fatom* (*get-atom l*)] **by** *blast*
moreover
then have $\neg$*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* $(f :: nat \Rightarrow partial\text{-}pred\text{-}denot)$
  assumes *model-c*: $\forall n.\ \neg falsifies_c\ (f\ n)\ C$
  assumes *fin-c*: *finite C*
  shows $eval_c$ *HFun* (*extend f*) *C*
unfolding $eval_c$*-def* **proof**
  fix *E*
  let *?G* = *extend f*
  let $?\sigma$ = *sub-of-denot E*

  from *fin-c* **have** *fin-c*σ: *finite* $(C \cdot_{ls}$ *sub-of-denot E*$)$ **by** *auto*
  have *groundc*σ: $ground_{ls}$ $(C \cdot_{ls}$ *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** $\forall n.\ \neg falsifies_c\ (f\ n)\ (C \cdot_{ls}\ ?\sigma)$ **using** *partial-equiv-subst* **by**
blast
  — Then from syntactic ground world to semantic ground world:
  then have $eval_c$ *HFun ?G* $(C \cdot_{ls}\ ?\sigma)$ **using** *groundc*σ *f-infpath fin-c*σ *extend-preserves-model2*[*of*
f $C \cdot_{ls}\ ?\sigma$] **by** *blast*
  — Then from semantic ground world to semantic FO world:
  then have $\forall E.\ \exists l \in (C \cdot_{ls}\ ?\sigma).\ eval_l\ E$ *HFun ?G l* **unfolding** $eval_c$*-def* **by**
auto
  then have $\exists l \in (C \cdot_{ls}\ ?\sigma).\ eval_l\ E$ *HFun ?G l* **by** *auto*
  then show $\exists l \in C.\ eval_l\ E$ *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* $(f :: nat \Rightarrow partial\text{-}pred\text{-}denot)$
  assumes *model-cs*: $\forall n.\ \neg falsifies_{cs}\ (f\ n)\ Cs$
  assumes *fin-cs*: *finite Cs*
  assumes *fin-c*: $\forall C \in Cs.$ *finite C*
  shows $eval_{cs}$ *HFun* (*extend f*) *Cs*
proof −
  let *?F* = *HFun*

  have $\forall C \in Cs.\ eval_c$ *?F* (*extend f*) *C*
    proof (*rule ballI*)
      fix *C*
      assume *asm*: $C \in Cs$
      then have $\forall n.\ \neg falsifies_c\ (f\ n)\ C$ **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*: *branch b* (*deeptree n*) ⟹ *length b* = *n*
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-from-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 $l{\in}C$
then have $\mathit{falsifies}_l\ (ds\ @\ d)\ l$ using $\mathit{assms}\ \mathit{longer\text{-}falsifies}_l$ by auto
} then show $?\mathit{thesis}$ using assms by auto
qed

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

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

lemma $\mathit{longer\text{-}falsifies}$:
assumes $\mathit{falsifies}_{cs}\ ds\ Cs$
shows $\mathit{falsifies}_{cs}\ (ds\ @\ d)\ Cs$
proof −
from assms **obtain** C **where** $C \in Cs \wedge \mathit{falsifies}_c\ ds\ C$ **by** auto
moreover
then have $\mathit{falsifies}_c\ (ds\ @\ d)\ C$ **using** $\mathit{longer\text{-}falsifies}_c[\mathit{of}\ C\ ds\ d]$ **by** blast
ultimately
show $?\mathit{thesis}$ **by** auto
qed

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

theorem $\mathit{herbrand}'$:
assumes openb: $\forall\, T.\ \exists\, G.\ \mathit{open\text{-}branch}\ G\ T\ Cs$
assumes $\mathit{finite\text{-}cs}$: $\mathit{finite}\ Cs\ \forall\, C{\in}Cs.\ \mathit{finite}\ C$
shows $\exists\, G.\ \mathit{eval}_{cs}\ \mathit{HFun}\ G\ Cs$
proof −
— Show T infinite:
let $?\mathit{tree} = \{G.\ \neg\mathit{falsifies}_{cs}\ G\ Cs\}$
let $?\mathit{undiag} = \mathit{length}$
let $?\mathit{diag} = (\lambda l.\ \mathit{SOME}\ b.\ \mathit{open\text{-}branch}\ b\ (\mathit{deeptree}\ l)\ Cs) :: \mathit{nat} \Rightarrow \mathit{partial\text{-}pred\text{-}denot}$

from openb **have** $\mathit{diag\text{-}open}$: $\forall\, l.\ \mathit{open\text{-}branch}\ (?\mathit{diag}\ l)\ (\mathit{deeptree}\ l)\ Cs$
using $\mathit{someI\text{-}ex}[\mathit{of}\ \lambda b.\ \mathit{open\text{-}branch}\ b\ (\mathit{deeptree}\ \text{-})\ Cs]$ **by** auto
then have $\forall\, n.\ ?\mathit{undiag}\ (?\mathit{diag}\ n) = n$ **using** $\mathit{branch\text{-}length}$ **by** auto
moreover
have $\forall\, n.\ (?\mathit{diag}\ n) \in ?\mathit{tree}$ **using** $\mathit{diag\text{-}open}$ **by** auto
ultimately
have $\neg\mathit{finite}\ ?\mathit{tree}$ **using** $\mathit{infinity}[\mathit{of}\ \text{-}\ \lambda n.\ \mathit{SOME}\ b.\ \mathit{open\text{-}branch}\ b\ (\text{-}\ n)\ Cs]$ **by** simp

— Get infinite path:
moreover
have $\forall$ *ds d*. $\neg$*falsifies*$_{cs}$ (*ds* @ *d*) *Cs* $\longrightarrow$ $\neg$*falsifies*$_{cs}$ *ds Cs*
using *longer-falsifies*[*of Cs*] **by** *blast*
then have ($\forall$ *ds d*. *ds* @ *d* $\in$ *?tree* $\longrightarrow$ *ds* $\in$ *?tree*) **by** *auto*
ultimately
have $\exists$ *c*. *wf-infpath c* $\wedge$ ($\forall$ *n*. *c n* $\in$ *?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-from-fatom* (*get-atom l*) < *length ds*
shows *falsifies*$_l$ *ds l*
proof –
let *?i* = *nat-from-fatom* (*get-atom l*)
from *assms* **have** *i-p*: *ground*$_l$ *l* $\wedge$ *?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* $\forall$ *C*$\in$*Cs*. *finite C*
assumes *unsat*: $\forall$ *G*. $\neg$*eval*$_{cs}$ *HFun G Cs*
shows $\exists$ *T*. $\forall$ *G*. *branch G T* $\longrightarrow$ *closed-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*: $\forall$ *G*. $\neg$*eval*$_{cs}$ *HFun G Cs*
assumes *finite-cs*: *finite Cs* $\forall$ *C*$\in$*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*) $\wedge$ *anyinternal T* (λ*p*. $\neg$ *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 [5] in the IsaFoL project [2]. It uses a similar theorem from the IsaFoR [8] project.

lemma *lifting*:
  assumes *fin*: $\mathit{finite}\ C \wedge \mathit{finite}\ D$
  assumes *apart*: $\mathit{vars}_{ls}\ C \cap \mathit{vars}_{ls}\ D = \{\}$
  assumes inst_1: $\mathit{instance\text{-}of}_{ls}\ C'\ C$
  assumes inst_2: $\mathit{instance\text{-}of}_{ls}\ D'\ D$
  assumes *appl*: $\mathit{applicable}\ C'\ D'\ L'\ M'\ \sigma$
  shows $\exists L\ M\ \tau.\ \mathit{applicable}\ C\ D\ L\ M\ \tau\ \wedge$
    $\mathit{instance\text{-}of}_{ls}\ (\mathit{resolution}\ C'\ D'\ L'\ M'\ \sigma)\ (\mathit{resolution}\ C\ D\ L\ M\ \tau)$
proof −
  let $?C'_1 = C' - L'$
  let $?D'_1 = D' - M'$

  from inst_1 **obtain** *lmbd* **where** *lmbd-p*: $C \cdot_{ls} \mathit{lmbd} = C'$ **unfolding** $\mathit{instance\text{-}of}_{ls}\text{-}\mathit{def}$ **by** *auto*
  from inst_2 **obtain** μ **where** μ*-p*: $D \cdot_{ls} \mu = D'$ **unfolding** $\mathit{instance\text{-}of}_{ls}\text{-}\mathit{def}$ **by** *auto*

  from μ*-p* *lmbd-p* *apart* **obtain** η **where** η*-p*: $C \cdot_{ls} \eta = C' \wedge D \cdot_{ls} \eta = D'$ **using** *merge-sub* **by** *force*

  from η*-p* **have** $\exists L \subseteq C.\ L \cdot_{ls} \eta = L' \wedge (C - L) \cdot_{ls} \eta = ?C'_1$ **using** *appl* *project-sub*[*of* $\eta\ C\ C'\ L'$] **unfolding** *applicable-def* **by** *auto*
  then obtain L **where** *L-p*: $L \subseteq C \wedge L \cdot_{ls} \eta = L' \wedge (C - L) \cdot_{ls} \eta = ?C'_1$ **by** *auto*
  let $?C_1 = C - L$

  from η*-p* **have** $\exists M \subseteq D.\ M \cdot_{ls} \eta = M' \wedge (D - M) \cdot_{ls} \eta = ?D'_1$ **using** *appl* *project-sub*[*of* $\eta\ D\ D'\ M'$] **unfolding** *applicable-def* **by** *auto*
  then obtain M **where** *M-p*: $M \subseteq D \wedge M \cdot_{ls} \eta = M' \wedge (D - M) \cdot_{ls} \eta = ?D'_1$ **by** *auto*
  let $?D_1 = D - M$

  from *appl* **have** $\mathit{mgu}_{ls}\ \sigma\ (L' \cup M'^C)$ **unfolding** *applicable-def* **by** *auto*
  then have $\mathit{mgu}_{ls}\ \sigma\ ((L \cdot_{ls} \eta) \cup (M \cdot_{ls} \eta)^C)$ **using** *L-p M-p* **by** *auto*
  then have $\mathit{mgu}_{ls}\ \sigma\ ((L \cup M^C) \cdot_{ls} \eta)$ **using** *compls-subls subls-union* **by** *auto*
  then have $\mathit{unifier}_{ls}\ \sigma\ ((L \cup M^C) \cdot_{ls} \eta)$ **unfolding** $\mathit{mgu}_{ls}\text{-}\mathit{def}$ **by** *auto*

then have $\eta\sigma uni$: *unifier*$_{ls}$ $(\eta \cdot \sigma)$ $(L \cup M^C)$
unfolding *unifier*$_{ls}$*-def* **using** *composition-conseq2l* **by** *auto*
then obtain τ **where** τ*-p*: *mgu*$_{ls}$ τ $(L \cup M^C)$ **using** *unification fin* **by** (*meson L-p M-p finite-UnI finite-imageI rev-finite-subset*)
then obtain φ **where** φ*-p*: $\tau \cdot \varphi = \eta \cdot \sigma$ **using** $\eta\sigma uni$ **unfolding** *mgu*$_{ls}$*-def* **by** *auto*

— Showing that we have the desired resolvent:
let *?E* $= ((C - L) \cup (D - M)) \cdot_{ls} \tau$
have *?E* $\cdot_{ls} \varphi = (?C_1 \cup ?D_1) \cdot_{ls} (\tau \cdot \varphi)$ **using** *subls-union composition-conseq2ls* **by** *auto*
also have ... $= (?C_1 \cup ?D_1) \cdot_{ls} (\eta \cdot \sigma)$ **using** φ*-p* **by** *auto*
also have ... $= ((?C_1 \cdot_{ls} \eta) \cup (?D_1 \cdot_{ls} \eta)) \cdot_{ls} \sigma$ **using** *subls-union composition-conseq2ls* **by** *auto*
also have ... $= (?C'_1 \cup ?D'_1) \cdot_{ls} \sigma$ **using** η*-p L-p M-p* **by** *auto*
finally have *?E* $\cdot_{ls} \varphi = ((C' - L') \cup (D' - M')) \cdot_{ls} \sigma$ **by** *auto*
then have *inst*: *instance-of*$_{ls}$ (*resolution* C' D' L' M' σ) (*resolution* C D L M τ)
unfolding *resolution-def instance-of*$_{ls}$*-def* **by** *blast*

— Showing that the resolution is applicable:
{
have $C' \neq \{\}$ **using** *appl* **unfolding** *applicable-def* **by** *auto*
then have $C \neq \{\}$ **using** η*-p* **by** *auto*
} moreover {
have $D' \neq \{\}$ **using** *appl* **unfolding** *applicable-def* **by** *auto*
then have $D \neq \{\}$ **using** η*-p* **by** *auto*
} moreover {
have $L' \neq \{\}$ **using** *appl* **unfolding** *applicable-def* **by** *auto*
then have $L \neq \{\}$ **using** *L-p* **by** *auto*
} moreover {
have $M' \neq \{\}$ **using** *appl* **unfolding** *applicable-def* **by** *auto*
then have $M \neq \{\}$ **using** *M-p* **by** *auto*
}
ultimately have *appll*: *applicable* C D L M τ
using *apart L-p M-p* τ*-p* **unfolding** *applicable-def* **by** *auto*

from *inst appll* **show** *?thesis* **by** *auto*
qed

17 Completeness

lemma *falsifies*$_g$*-empty*:
assumes *falsifies*$_g$ [] C
shows $C = \{\}$
proof −
have $\forall l \in C.$ *False*
proof
fix l

assume $l \in C$
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 $\wedge$ *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'*: $l_1 \in C_1'$
assumes l_2*C1'*: $l_2 \in C_1'$
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-from-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-from-fatom* (*get-atom* l_2)
from *falsif* **have** *falsifies*$_l$ G l_2 **using** l_2*C1'* **by** *auto*
then have G ! *?i2* = ($\neg$*sign* l_2) **unfolding** *falsifies*$_l$*-def* **by** *meson*
then have G ! *?i1* = ($\neg$*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-from-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-from-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-from-fatom* (*get-atom* l) = *length B*
assumes *other*: $lo \in C_1{'}$ $lo \neq l$
shows $falsifies_l$ *B lo*
proof −
let *?i* = *nat-from-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-from-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*: $lo \neq l^c$ **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-from-fatom* (*get-atom lo*) ≠ *nat-from-fatom* (*get-atom l*)
using *nat-from-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-from-fatom* (*get-atom lo*) ≠ *length B* **using** *l-p* **by** *auto*
ultimately
— So their numbers are in the range of B:
have *nat-from-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′*:
shows *closed-tree T Cs* $\Longrightarrow \forall$ *C*$\in$*Cs*. *finite C* $\Longrightarrow \exists$ *Cs′*. *resolution-deriv Cs Cs′* $\wedge$ {} $\in$ *Cs′*
proof (*induction T arbitrary*: *Cs rule*: *measure-induct-rule*[*of treesize*])
fix *T*::*tree*
fix *Cs* :: *fterm clause set*
assume *ih*: ($\bigwedge$ *T′ Cs*. *treesize T′* < *treesize T* $\Longrightarrow$ *closed-tree T′ Cs* $\Longrightarrow \forall$ *C*$\in$*Cs*. *finite C* $\Longrightarrow$
$\exists$ *Cs′*. *resolution-deriv Cs Cs′* $\wedge$ {} $\in$ *Cs′*)
assume *clo*: *closed-tree T Cs*
assume *finite-Cs*: $\forall$ *C*$\in$*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-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_1o **where** C_1o*-p*: C_1o $\in$ *Cs* $\wedge$ *falsifies*$_c$ *?B*$_1$ C_1o **using** *b-p clo* **unfolding** *closed-tree-def* **by** *metis*
obtain C_2o **where** C_2o*-p*: C_2o $\in$ *Cs* $\wedge$ *falsifies*$_c$ *?B*$_2$ C_2o **using** *b-p clo* **unfolding** *closed-tree-def* **by** *metis*

— Standardizing the clauses apart:
let *?C*$_1$ = *std*$_1$ C_1o
let *?C*$_2$ = *std*$_2$ C_2o
have C_1*-p*: *falsifies*$_c$ *?B*$_1$ *?C*$_1$ **using** *std*$_1$*-falsifies* C_1o*-p* **by** *auto*
have C_2*-p*: *falsifies*$_c$ *?B*$_2$ *?C*$_2$ **using** *std*$_2$*-falsifies* C_2o*-p* **by** *auto*

have *fin*: *finite* $?C_1$ $\wedge$ *finite* $?C_2$ **using** C_1o-*p* C_2o-*p* *finite-Cs* **by** *auto*

— We go down to the ground world.

— Finding the falsifying ground instance C_1' of $C_1 o \cdot_{ls} (\lambda x.\ \varepsilon\ (''1'' @ x))$, 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_1o **using** C_1o-*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-from-fatom* (*get-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-from-fatom* (*get-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 $C_2 o \cdot_{ls} (\lambda x.\ \varepsilon\ (''2'' @ x))$, 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-from-fatom* (*get-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-from-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-from-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 $C_1 o \cdot_{ls}$ ($\lambda x.\ \varepsilon$ (''1'' @ x)) and $C_2 o \cdot_{ls}$ ($\lambda x.\ \varepsilon$ (''2'' @ x)):
then obtain L_1 L_2 τ **where** $L_1L_2\tau$*-p*: *applicable ?C*$_1$ *?C*$_2$ L_1 L_2 τ $\wedge$ *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* $\cup$ $\{?C_1,\ ?C_2,\ C\}$ **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 $((C_1{}' - \{l_1\}) \cup (C_2{}' - \{l_2\}))$ **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* $L_1L_2\tau$-*p* **by** *auto*

have T''-*smaller*: *treesize* T'' < *treesize* T **using** *treezise-delete* T''-*p* *b-p* **by** *auto*
have T''-*bran*: *anybranch* T'' (λb. *closed-branch* b T'' *CsNext*)
proof (*rule allI*; *rule impI*)
fix b
assume *br*: *branch* b T''
from *br* **have** $b = B \vee$ *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'' (λb. $falsifies_{cs}$ 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* (λG. $falsifies_{cs}$ G *CsNext*) [] T'' **by** *auto*
have T'-*smaller*: *treesize* T' < *treesize* T **using** *treesize-cutoff*[*of* λG. $falsifies_{cs}$ G *CsNext* [] T''] T''-*smaller* **unfolding** T'-*p* **by** *auto*

from T''-*bran2* **have** *anybranch* T' (λb. $falsifies_{cs}$ b *CsNext*) **using** *cutoff-branch*[*of* T'' λb. $falsifies_{cs}$ b *CsNext*] T'-*p* **by** *auto*
then have T'-*bran*: *anybranch* T' (λb. *closed-branch* b T' *CsNext*) **by** *auto*
have T'-*intr*: *anyinternal* T' (λp. $\neg$$falsifies_{cs}$ p *CsNext*) **using** T'-*p* *cutoff-internal*[*of* T'' λb. $falsifies_{cs}$ 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*: $\forall\, C{\in}CsNext.$ *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** $\exists\, Cs''.$ *resolution-deriv CsNext* $Cs'' \wedge \{\} \in Cs''$ **using** *ih*[*of T′ CsNext*] *finite-CsNext* **by** *blast*
then obtain Cs'' **where** *Cs″-p*: *resolution-deriv CsNext* $Cs'' \wedge \{\} \in Cs''$ **by** *auto*
moreover
{ — Proving that we can actually derive the new clausal form:
have *resolution-step Cs* $(Cs \cup \{?C_1\})$ **using** *std*$_1$*-renames standardize-apart* C_1*o-p* **by** (*metis Un-insert-right*)
moreover
have *resolution-step* $(Cs \cup \{?C_1\})$ $(Cs \cup \{?C_1\} \cup \{?C_2\})$ **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* $(Cs \cup \{?C_1\})$ $(Cs \cup \{?C_1, ?C_2\})$ **by** (*simp add*: *insert-commute*)
moreover
then have *resolution-step* $(Cs \cup \{?C_1, ?C_2\})$ $(Cs \cup \{?C_1, ?C_2\} \cup \{C\})$
using $L_1 L_2 \tau$*-p resolution-rule*[*of* $?C_1$ $Cs \cup \{?C_1, ?C_2\}$ $?C_2$ L_1 L_2 τ] **using** *C-p* **by** *auto*
then have *resolution-step* $(Cs \cup \{?C_1, ?C_2\})$ *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*: $\forall\, (F{::}hterm\ fun\text{-}denot)\ (G{::}hterm\ pred\text{-}denot)\ .\ \neg eval_{cs}\ F\ G\ Cs$
shows $\exists\, Cs'.$ *resolution-deriv Cs* $Cs' \wedge \{\} \in Cs'$
proof −
from *unsat* **have** $\forall\, (G{::}hterm\ pred\text{-}denot)\ .\ \neg eval_{cs}\ HFun\ G\ Cs$ **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

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 $Pos\ ''equals''$
$\quad [Fun\ ''add''[Fun\ ''mul''[Var\ ''y'',Var\ ''y''],\ Fun\ ''one''[]],Var\ ''x'']$

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

fun G_{nat} :: *nat pred-denot* **where**
$\quad G_{nat}\ p\ [x,y] =$
$\quad\quad (if\ p = ''less'' \wedge x < y\ then\ True\ else$
$\quad\quad\ if\ p = ''greater'' \wedge x > y\ then\ True\ else$
$\quad\quad\ if\ p = ''equals'' \wedge x = y\ then\ True\ else\ False)$
$|\ G_{nat}\ p\ us = False$

fun E_{nat} :: *nat var-denot* **where**
$\quad E_{nat}\ x =$
$\quad\quad (if\ x = ''x''\ then\ 26\ else$
$\quad\quad\ if\ x = ''y''\ then\ 5\ else\ 0)$

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*: $unifier_{ls}\ \varepsilon\ L \Longrightarrow mgu_{ls}\ \varepsilon\ L$
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*: $unifier_{ls}\ \sigma\ \{l\}$
unfolding $unifier_{ls}$*-def* **by** *auto*

theorem *resolution-rule'*:
    $C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow applicable\ C_1\ C_2\ L_1\ L_2\ \sigma$
  $\Longrightarrow C = \{resolution\ C_1\ C_2\ L_1\ L_2\ \sigma\}$
  $\Longrightarrow resolution\text{-}step\ Cs\ (Cs \cup C)$
  using *resolution-rule* **by** *auto*

lemma *resolution-example1*:
  $resolution\text{-}deriv\ \{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}$
      $\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\},\{PP\},\{\}\}$
proof –
  have *resolution-step*
      $\{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}$
      $(\{\{NP,PQ\},\{NQ\},\{PP,PQ\}\} \cup \{\{NP\}\})$
    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**
  $mguPaaPax = (\lambda x.\ if\ x = ''x''\ then\ Fun\ ''a''\ []\ else\ Var\ x)$

lemma *mguPaaPax-mgu*: $mgu_{ls}\ mguPaaPax\ \{Paa,Pax\}$
proof −
  let $?\sigma = \lambda x.\ if\ x = ''x''\ then\ Fun\ ''a''\ []\ else\ Var\ x$
  have *a*: $unifier_{ls}\ (\lambda x.\ if\ x = ''x''\ then\ Fun\ ''a''\ []\ else\ Var\ x)\ \{Paa,Pax\}$ **unfolding** *Paa-def Pax-def* $unifier_{ls}$*-def* **by** *auto*
  have *b*: $\forall u.\ unifier_{ls}\ u\ \{Paa,Pax\} \longrightarrow (\exists i.\ u = ?\sigma \cdot i)$
    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 $?\sigma \cdot u = u$
        proof
          fix *x*
          {
            assume $x=''x''$
            moreover
            have $(?\sigma \cdot u)\ ''x'' = Fun\ ''a''\ []$ **unfolding** *composition-def* **by** *auto*
            ultimately have $(?\sigma \cdot u)\ x = u\ x$ **using** *uuu* **by** *auto*
          }
          moreover
          {
            assume $x\neq''x''$
            then have $(?\sigma \cdot u)\ x = (\varepsilon\ x) \cdot_t u$ **unfolding** *composition-def* **by** *auto*
            then have $(?\sigma \cdot u)\ x = u\ x$ **by** *auto*
          }
          ultimately show $(?\sigma \cdot u)\ x = u\ x$ **by** *auto*
        qed
      then have $\exists i.\ ?\sigma \cdot i = u$ **by** *auto*
      then show $\exists i.\ u = ?\sigma \cdot i$ **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 *ref-sound*:
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** *lsound-derivation* **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

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

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

end

References

[1] M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 3rd edition, 2012.

[2] J. C. Blanchette, M. Fleury, A. Schlichtkrull, and D. Traytel. IsaFoL: Isabelle Formalization of Logic. https://bitbucket.org/jasmin_blanchette/isafol.

[3] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, Inc., Orlando, FL, USA, 1st edition, 1973.

[4] A. Leitsch. *The Resolution Calculus.* Texts in theoretical computer science. Springer, 1997.

[5] A. Schlichtkrull. Formalization of first-order unordered resolution. https://bitbucket.org/jasmin_blanchette/isafol/src/master/Unordered_Resolution/.

[6] A. Schlichtkrull. Formalization of resolution calculus in Isabelle. Msc thesis, Technical University of Denmark, 2015. https://people.compute.dtu.dk/andschl/Thesis.pdf.

[7] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In *ITP 2016*, volume 9807 of *LNCS*. Springer, 2016.

[8] C. Sternagel and R. Thiemann. An Isabelle/HOL formalization of rewriting for certified termination analysis. http://cl-informatik.uibk.ac.at/software/ceta/.