The Resolution Calculus for First-Order Logic

Schlichtkrull, Anders

Published in:
Archive of Formal Proofs

Publication date:
2016

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

Link back to DTU Orbit

Citation (APA):
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 2
  1.1 Ground .................................................. 3
  1.2 Auxiliary ................................................... 3
  1.3 Conversions ............................................... 4
    1.3.1 Conversions - Terms and Herbrand Terms ........ 4
    1.3.2 Conversions - Literals and Herbrand Literals .... 5
    1.3.3 Conversions - Atoms and Herbrand Atoms ....... 5
  1.4 Enumerations ............................................. 6
    1.4.1 Enumerating Strings ................................. 6
    1.4.2 Enumerating Herbrand Atoms .................... 7
    1.4.3 Enumerating Ground Atoms ...................... 8

2 Trees 9
  2.1 Sizes ................................................... 9
  2.2 Paths .................................................... 9
  2.3 Branches ................................................ 11
  2.4 Internal Paths ......................................... 13
  2.5 Deleting Nodes ......................................... 15
3 Possibly Infinite Trees 22
  3.1 Infinite Paths 23

4 König’s Lemma 24

5 More Terms and Literals 25

6Clauses 26

7 Semantics 27
  7.1 Semantics of Ground Terms 28

8 Substitutions 28
  8.1 The Empty Substitution 29
  8.2 Substitutions and Ground Terms 30
  8.3 Composition 31
  8.4 Merging substitutions 33
  8.5 Standardizing apart 35

9 Unifiers 36
  9.1 Most General Unifiers 38

10 Resolution 39

11 Soundness 40

12 Herbrand Interpretations 43

13 Partial Interpretations 44

14 Semantic Trees 48

15 Herbrand’s Theorem 48

16 Lifting Lemma 54

17 Completeness 55

18 Examples 63

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 \texttt{fterm} = 
\begin{align*}
\text{Fun} & \quad \text{fun-sym} \ (\text{get-sub-terms: fterm list}) \\
\mid & \quad \text{Var} \quad \text{var-sym}
\end{align*}

datatype \texttt{hterm} = H\text{Fun} \quad \text{fun-sym} \ \text{hterm list} \quad \text{— Herbrand terms defined as in Berghofer’s FOL-Fitting}

type-synonym \ 't \ atom = \text{pred-sym} \ast \ 't \ list

datatype \ 't \ literal = 
\begin{align*}
\text{Pos} & \quad \text{get-pred: pred-sym} \ (\text{get-terms: 't list}) \\
\mid & \quad \text{Neg} \quad \text{get-pred: pred-sym} \ (\text{get-terms: 't list})
\end{align*}

fun \text{get-atom} :: \ 't \ literal \Rightarrow \ 't \ atom \ where
\begin{align*}
\text{get-atom} \ (\text{Pos} \ p \ ts) & = (p, ts) \\
\mid & \quad \text{get-atom} \ (\text{Neg} \ p \ ts) = (p, ts)
\end{align*}

1.1 Ground

fun \text{ground} :: \ fterm \Rightarrow \ \text{bool} \ where
\begin{align*}
\text{ground} \ (\text{Var} \ x) & \leftrightarrow \text{False} \\
\mid & \quad \text{ground} \ (\text{Fun} \ f \ ts) \leftrightarrow (\forall t \in \text{set} \ ts. \ \text{ground} \ t)
\end{align*}

abbreviation \text{ground}_{\ast} :: \ fterm \ list \Rightarrow \ \text{bool} \ where
\begin{align*}
\text{ground}_{\ast} \ ts & \equiv (\forall t \in \text{set} \ ts. \ \text{ground} \ t)
\end{align*}

abbreviation \text{ground}_{\ast} :: \ fterm \ literal \Rightarrow \ \text{bool} \ where
\begin{align*}
\text{ground}_{\ast} \ l & \equiv \text{ground}_{\ast} \ (\text{get-terms} \ l)
\end{align*}

definition \text{ground-fatoms} :: \ fterm \ atom \ set \ where
\begin{align*}
\text{ground-fatoms} & \equiv \{a. \ \text{ground}_{\ast} \ (\text{snd} \ a)\}
\end{align*}

lemma \text{ground}_{\ast} \text{-ground-fatom}: \text{ground}_{\ast} \ l \Longrightarrow \text{get-atom} \ l \in \text{ground-fatoms}

unfolding \text{ground-fatoms-def by} \ (\text{induction} \ l) \ \text{auto}

1.2 Auxiliary

lemma \text{infinity}:
\begin{align*}
\text{assumes inj} : \forall n :: \text{nat.} \ \text{undiago} \ (\text{diago} \ n) & = n \\
\text{assumes all-tree} : \forall n :: \text{nat.} \ (\text{diago} \ n) \in S \\
\text{shows} & \quad \neg \text{finite} \ S
\end{align*}

proof —
\begin{align*}
\text{from inj all-tree have} \ & \forall n. \ n = \text{undiago} \ (\text{diago} \ n) \land (\text{diago} \ n) \in S \text{ by auto} \\
\text{then have} \ & \forall n. \ \exists ds. \ n = \text{undiago} \ ds \land ds \in S \text{ by auto}
\end{align*}
then have \( \text{undia}_S \colon S = (\text{UNIV} :: \text{nat set}) \) by auto
then show \( \neg \text{finite } S \) by (metis finite-imageI infinite-UNIV-nat)
qed

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

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

1.3 Conversions

1.3.1 Conversions - Terms and Herbrand Terms

fun fterm-of-hterm :: hterm \( \Rightarrow \) fterm where
fterm-of-hterm (HFun p ts) = Fun p (map fterm-of-hterm ts)

definition fterms-of-hterms :: hterm list \( \Rightarrow \) fterm list where
fterms-of-hterms ts \( \equiv \) map fterm-of-hterm ts

fun hterm-of-fterm :: fterm \( \Rightarrow \) hterm where
hterm-of-fterm (Fun p ts) = HFun p (map hterm-of-fterm ts)

definition hterms-of-fterms :: fterm list \( \Rightarrow \) hterm list where
hterms-of-fterms ts \( \equiv \) 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 \( \Rightarrow \) fterm-of-hterm (hterm-of-fterm t) = t
by (induction t) (auto simp add: map-idI)

lemma [simp]: ground ts \( \Rightarrow \) 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-inv
    by (metis (mono-tags, lifting) mem-Collect-eq)
    next
    have ∀l. ∃l'. ground l l' ∧ l = hlit-of-flit l'
      using ground-flit-of-hlit hlit-of-flit-flit-of-hlit by metis
    then show hlit-of-flit ' {l. ground l l} = UNIV by auto
  qed

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

1.3.3 Conversions - Atoms and Herbrand Atoms

fun fatom-of-hatom :: hterm atom ⇒ fterm atom where
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_\langle\;\rangle (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_\langle\;\rangle (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_\langle\;\rangle (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 hatom-of-fatom-bij: bij-betw fatom-of-hatom ground-fatoms UNIV
  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_\langle\;\rangle (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_\langle\;\rangle (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-fatoms 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 = (λn. fatom-of-hatom (hatom-from-nat n))

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

theorem diag-undiag-fatom[simp]: groundts 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: groundts (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 bools as directions in a binary tree
hide-const (open) Left Right
type-synonym dir = bool
definition Left :: bool where Left = True
definition Right :: bool where Right = False
declare Left-def [simp]
declare Right-def [simp]
datatype tree =
  Leaf
| Branching (ltree: tree) (rtree: tree)

2.1 Sizes
fun treesize :: tree ⇒ 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: \( \text{path}(a \# ds) \ T \to (\exists l \ r. \ T = \text{Branching} \ l \ r) \)
by (cases T) (auto simp add: path-inv-Leaf)

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

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

lemma path-inv-Branching:
\[ \text{path} \ p \ (\text{Branching} \ l \ r) \longleftrightarrow (p=\[] \lor (\exists a \ p'. \ p=a\#p' \land (a \to \text{path} \ p' \ l) \land (\neg a \to \text{path} \ p' \ r)) \ (\text{is} \ L \longleftrightarrow \text{R}) \]
proof
assume \( L \) then show \( R \) by (induction p) auto
next
assume \( r : \text{R} \) then show \( L \) proof
assume \( p = \[] \) then show \( L \) by auto
next
assume \( \exists a \ p'. \ p=a\#p' \land (a \to \text{path} \ p' \ l) \land (\neg a \to \text{path} \ p' \ r) \)
then obtain \( a \ p' \) where \( p=a\#p' \land (a \to \text{path} \ p' \ l) \land (\neg a \to \text{path} \ p' \ r) \) by auto
then show \( L \) by (cases a) auto
qed
qed

lemma path-prefix: \( \text{path}(ds1@ds2) \ T \implies \text{path} \ ds1 \ T \)
proof (induction ds1 arbitrary: T)
case (Cons a ds1)
then have \( \exists l \ r. \ T = \text{Branching} \ l \ r \) using path-inv-Leaf by (cases T) auto
then obtain \( l \ r \) where \( p-lr: \ T = \text{Branching} \ l \ r \) by auto
show \( ?c \)
proof (cases a)
assume \( \text{atrue: a} \)
then have \( \text{path} \ ((ds1) @ ds2) \ l \) using p-lr Cons(2) path-inv-Branching by auto
then have \( \text{path} \ ds1 \ l \) using Cons(1) by auto
then show \( \text{path} \ (a \# ds1) \ T \) using p-lr atrue by auto
next
assume \( a\text{false: } \neg a \)
then have \( \text{path} \ ((ds1) @ ds2) \ r \) using p-lr Cons(2) path-inv-Branching by auto
then have \( \text{path} \ ds1 \ r \) using Cons(1) by auto
then show \( \text{path} \ (a \# ds1) \ T \) using p-lr afalse by auto
qed
next

10
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: 3 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’. b=a # b')) ∧ (∃ a b’ l) ∧ (∃ a 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’ l. b=a # b’) by auto \}
then have \( \forall b. \) branch \( b \) \( T \rightarrow \neg (\exists l r. \) branch \( b \) \( (\text{Branching } l r)) \)
using \text{branch-inv-Branching} by auto
then have \( T=\text{Leaf} \) using \text{has-branch[of } T] by (metis \text{branch.elims}(2))

ultimately show \( T = \text{Leaf} \leftrightarrow (\forall b. \) branch \( b \) \( T \rightarrow b = []) \) by auto

qed

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

lemma \text{Branching-Leaf-Leaf-Tree}:
\( T = \text{Branching } T_1 \ T_2 \rightarrow (\exists B. \) branch \( (B@[True]) T \land \text{branch } (B@[False]) T) \)
proof (induction \( T \) arbitrary: \( T_1 \ T_2 \))
case \text{Leaf} then show \?case by auto
next
case \( (\text{Branching } T_1' \ T_2') \)
\{ 
assume \( T_1'=\text{Leaf} \land T_2'=\text{Leaf} \)
then have \( \text{branch } ([] @ [\text{True}]) (\text{Branching } T_1' \ T_2') \land \text{branch } ([] @ [\text{False}]) (\text{Branching } T_1' \ T_2') \) by auto
then have \?case by metis
\}
moreover
\{ 
fix \( T_11 \ T_12 \)
assume \( T_1' = \text{Branching } T_11 \ T_12 \)
then obtain \( B \) where \( \text{branch } (B@[\text{True}]) T_1' \land \text{branch } (B@[\text{False}]) T_1' \) using \text{Branching} by blast
then have \( \text{branch } ([\text{True}] @ B) @ [\text{True}] (\text{Branching } T_1' \ T_2') \land \text{branch } ([\text{True}] @ B) @ [\text{False}] (\text{Branching } T_1' \ T_2') \) by auto
then have \?case by blast
\}
moreover
\{ 
fix \( T_11 \ T_12 \)
assume \( T_2' = \text{Branching } T_11 \ T_12 \)
then obtain \( B \) where \( \text{branch } (B@[\text{True}]) T_2' \land \text{branch } (B@[\text{False}]) T_2' \) using \text{Branching} by blast
\}
then have \( \text{branch} \left( \left[ \text{False} \right] @ B \right) @ \left[ \text{True} \right] \) (Branching \( T_1' \) \( T_2' \)) \land \text{branch} \left( \left[ \text{False} \right] @ B \right) @ \left[ \text{False} \right] \) (Branching \( T_1' \) \( T_2' \)) by auto

then have \?case by blast

ultimately show \?case using tree.exhaust by blast

qed

2.4 Internal Paths

fun internal :: \( \text{dir list} \Rightarrow \text{tree} \Rightarrow \text{bool} \)
where
\( \text{internal} [] \) (Branching \( l \) \( r \)) \( \leftarrow \) True
\| \text{internal} (d\#ds) (Branching \( l \) \( r \)) \( \leftarrow \) (if \( d \) then internal \( ds \) \( l \) else internal \( ds \) \( r \))
\| internal - - \( \leftarrow \) False

lemma internal-inv-Leaf: \( \neg \text{internal} \ b \ \text{Leaf} \) using internal.simps by blast

lemma internal-inv-Branching-Left:
\( \text{internal} \ (\text{Left}\#b) \) (Branching \( l \) \( r \)) \( \leftarrow \) internal \( b \) \( l \) by auto

lemma internal-inv-Branching-Right:
\( \text{internal} \ (\text{Right}\#b) \) (Branching \( l \) \( r \)) \( \leftarrow \) internal \( b \) \( r \)
by auto

lemma internal-inv-Branching:
\( \text{internal} \ p \) (Branching \( l \) \( r \)) \( \leftarrow \) (\( p=[] \lor (\exists a \ p'. \ p=a\#p'\land (a \rightarrow \text{internal} \ p' \ l) \land (\neg a \rightarrow \text{internal} \ p' \ r)) \) (is \?L \( \leftarrow \?R \))

proof
\( \text{assume} \?L \text{ then show } \?R \text{ by } (\text{metis internal.simps(2) neq-Nil-conv}) \)

next
\( \text{assume } r: \?R \)
then show \?L
proof
\( \text{assume } p = [] \text{ then show } \?L \text{ by auto} \)

next
\( \text{assume } \exists a \ p'. \ p=a\#p'\land (a \rightarrow \text{internal} \ p' \ l) \land (\neg a \rightarrow \text{internal} \ p' \ r) \)
then obtain \( a \ p' \) where \( p=a\#p'\land (a \rightarrow \text{internal} \ p' \ l) \land (\neg a \rightarrow \text{internal} \ p' \ r) \) by auto
then show \?L by (cases \( a \)) auto
qed

qed

lemma internal-is-path:
\( \text{internal} \ ds \ T = \Rightarrow \text{path} \ ds \ T \)

proof (induction \( T \) arbitrary: \( ds \))
\text{case } \text{Leaf}
then have \( \text{False} \) using internal-inv-Leaf by auto
then show \?case by auto
next
\text{case } (\text{Branching} \ T_1 \ T_2)
then obtain \( a \) \( b \) where \( ds - p : ds - [] \lor ds = a \# b \land (a \rightarrow \text{internal } b \ T_1) \land \) 
\( \neg a \rightarrow \text{internal } b \ T_2 \) using \( \text{internal-inv-Branching} \) by \( \text{blast} \) 
then have \( ds = [] \lor (a \rightarrow \text{path } b \ T_1) \land (\neg a \rightarrow \text{path } b \ T_2) \) using \( \text{Branching} \) 
by \( \text{auto} \) 
then show ?case using \( ds - p \) by (cases \( a \)) \( \text{auto} \) 
qed

lemma \( \text{internal-prefix} : \text{internal } (ds1 \cdot@ ds2 \cdot@ [d]) \ T \rightarrow \text{internal } ds1 \ T \) 
proof (induction \( ds1 \) arbitrary: \( T \)) 
  \begin{align*} 
  & \text{case } (\text{Cons} a \ ds1) \nonumber \\
  & \text{then have } \exists l \ r. \ T = \text{Branching } l \ r \text{ using } \text{internal-inv-Leaf} \text{ by } \text{(cases } T \text{)} \text{ auto} \\
  & \text{then obtain } l \ r \text{ where } p-lr:: T = \text{Branching } l \ r \text{ by } \text{auto} \\
  & \text{show } ?\text{case} \\
  & \quad \text{proof } \text{(cases } a \text{)} \\
  & \quad \quad \text{assume } a\text{true}: a \\
  & \quad \quad \text{then have } \text{internal } ((ds1 \cdot@ ds2 \cdot@ [d]) \ l \text{ using } p-lr\text{ Cons}(2) \text{ internal-inv-Branching by } \text{auto} \\
  & \quad \quad \text{then have } \text{internal } ds1 \ l \text{ using } \text{Cons}(1) \text{ by } \text{auto} \\
  & \quad \quad \text{then show } \text{internal } (a \# ds1) \ T \text{ using } p-lr \text{ atrue by } \text{auto} \\
  & \quad \text{next} \\
  & \quad \quad \text{assume } a\text{false}: \neg a \\
  & \quad \quad \text{then have } \text{internal } ((ds1 \cdot@ ds2 \cdot@ [d]) \ r \text{ using } p-lr\text{ Cons}(2) \text{ internal-inv-Branching by } \text{auto} \\
  & \quad \quad \text{then have } \text{internal } ds1 \ r \text{ using } \text{Cons}(1) \text{ by } \text{auto} \\
  & \quad \quad \text{then show } \text{internal } (a \# ds1) \ T \text{ using } p-lr \text{ afalse by } \text{auto} \\
  & \text{qed} \\
  \end{align*} 
next 
\text{case } (\text{Nil}) 
\text{then have } \exists l \ r. \ T = \text{Branching } l \ r \text{ using } \text{internal-inv-Leaf} \text{ by } \text{(cases } T \text{)} \text{ auto} \\
\text{then show } ?\text{case} \text{ by } \text{auto} \\
\text{qed}

lemma \( \text{internal-branch} : \text{branch } (ds1 \cdot@ ds2 \cdot@ [d]) \ T \rightarrow \text{internal } ds1 \ T \) 
proof (induction \( ds1 \) arbitrary: \( T \)) 
\begin{align*} 
  & \text{case } (\text{Cons} a \ ds1) \nonumber \\
  & \text{then have } \exists l \ r. \ T = \text{Branching } l \ r \text{ using } \text{branch-inv-Leaf} \text{ by } \text{(cases } T \text{)} \text{ auto} \\
  & \text{then obtain } l \ r \text{ where } p-lr:: T = \text{Branching } l \ r \text{ by } \text{auto} \\
  & \text{show } ?\text{case} \\
  & \quad \text{proof } \text{(cases } a \text{)} \\
  & \quad \quad \text{assume } a\text{true}: a \\
  & \quad \quad \text{then have } \text{branch } (ds1 \cdot@ ds2 \cdot@ [d]) \ l \text{ using } p-lr\text{ Cons}(2) \text{ branch-inv-Branching by } \text{auto} \\
  & \quad \quad \text{then have } \text{internal } ds1 \ l \text{ using } \text{Cons}(1) \text{ by } \text{auto} \\
  & \quad \quad \text{then show } \text{internal } (a \# ds1) \ T \text{ using } p-lr \text{ atrue by } \text{auto} \\
  & \quad \text{next} \\
  & \quad \quad \text{assume } a\text{false}: \neg a \\
  & \quad \quad \text{then have } \text{branch } ((ds1 \cdot@ ds2 \cdot@ [d]) \ r \text{ using } p-lr\text{ Cons}(2) \text{ branch-inv-Branching by } \text{auto} \\
  & \text{qed} \\
\end{align*}
then have \texttt{internal ds\ r using Cons(1) by auto}
then show \texttt{internal (a \# ds1) T using p-r \# afalse by auto}
qed

next

case (\texttt{Nil})
then have \exists l. r. \texttt{T = Branching l r using branch-inv-Leaf by \texttt{(cases T) auto}}
then show \texttt{?case by auto}
qed

fun \texttt{parent :: dir list \Rightarrow dir list where}
parent ds = \texttt{tl ds}

2.5 Deleting Nodes

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

lemma \texttt{delete-Leaf}: \texttt{delete T Leaf = Leaf by \texttt{(cases T) auto}}

lemma \texttt{path-delete}: \texttt{path p (delete ds T) \Rightarrow path p T}
proof (\texttt{induction p arbitrary: T ds})
case Nil
then show \texttt{?case by simp}
next
case (\texttt{Cons a p})
then obtain b ds\’ where \texttt{bds\’-p: ds=b\#ds\’ by \texttt{(cases ds) auto}}

have \exists dT_1 dT_2. \texttt{delete ds T = Branching dT_1 dT_2 using Cons path-inv-Cons by auto}
then obtain dT_1 dT_2 where \texttt{delete ds T = Branching dT_1 dT_2 by auto}

then have \exists T_1 T_2. \texttt{T=Branching T_1 T_2}
by (\texttt{cases T}; \texttt{cases ds) auto}
then obtain T_1 T_2 where \texttt{T1T2-p: T=Branching T_1 T_2 by auto}

{ 
assume a-p: a
assume b-p: \neg b
have \texttt{path (a \# p) (delete ds T) using Cons by} –
then have \texttt{path (a \# p) (Branching (T_1) (delete ds\’ T_2)) using b-p bds\’-p T1T2-p by auto}
then have \texttt{path p T1 using a-p by auto}
then have \texttt{?case using T1T2-p a-p by auto}
}
moreover
\{ 
  assume \( a \)-p: \textcolor{red}{\neg}a 
  assume \( b \)-p: \textcolor{red}{b} 
  have \( \text{path} \ (a \# p) \ (\text{delete} \ ds \ T) \ \text{using} \ \text{Cons} \ \text{by} \ - \)  
  then have \( \text{path} \ (a \# p) \ (\text{Branching} \ (\text{delete} \ ds' \ T1) \ T2) \ \text{using} \ \text{b-p} \ \text{bds}'-p \) \ 
  \( T1T2\)-p \ \text{by} \ \text{auto} \ 
  then have \( \text{path} \ p \ T2 \ \text{using} \ \text{a-p} \ \text{by} \ \text{auto} \)  
}\}

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

moreover 
\{ 
  assume \( a \)-p: \textcolor{red}{\neg}a 
  assume \( b \)-p: \textcolor{red}{\neg}b 
  have \( \text{path} \ (a \# p) \ (\text{delete} \ ds \ T) \ \text{using} \ \text{Cons} \ \text{by} \ - \)  
  then have \( \text{path} \ (a \# p) \ (\text{Branching} \ T1 \ (\text{delete} \ ds' \ T2)) \ \text{using} \ \text{b-p} \ \text{bds}'-p \) \ 
  \( T1T2\)-p \ \text{by} \ \text{auto} \ 
  then have \( \text{path} \ p \ (\text{delete} \ ds' \ T2) \ \text{using} \ \text{a-p} \ \text{by} \ \text{auto} \)  
  then have \( \text{path} \ p \ T2 \ \text{using} \ \text{Cons} \ \text{by} \ \text{auto} \)  
  then have \( ?\case \ \text{using} \ T1T2\)-p \ a-p \ \text{by} \ \text{auto} \)  
\}

ultimately show \( ?\case \ \text{by} \ \text{blast} \)
qed

\textbf{lemma} branch-delete: \( \text{branch} \ p \ (\text{delete} \ ds \ T) \Rightarrow \text{branch} \ p \ T \lor p=ds \)

\textbf{proof} \ (\text{induction} \ p \ \text{arbitrary}: \ T \ ds) 
\textbf{case} Nil 
then have \( \text{delete} \ ds \ T = \text{Leaf} \ \text{by} \ (\text{cases} \ \text{delete} \ ds \ T) \ \text{auto} \)  
then have \( ds = [] \lor T = \text{Leaf} \ \text{using} \ \text{delete.elims} \ \text{by} \ \text{blast} \)  
then show \( ?\case \ \text{by} \ \text{auto} \)
\textbf{next} 
\textbf{case} (\text{Cons} \ a \ p) 
then obtain \( b \ ds' \ \text{where} \ \text{bds}'\)-p: \( ds=b\#ds' \ \text{by} \ (\text{cases} \ ds) \ \text{auto} \)

\textbf{have} \( \exists \ dT1 \ dT2. \ \text{delete} \ ds \ T = \text{Branching} \ dT1 \ dT2 \ \text{using} \ \text{Cons} \ \text{path-inv-Cons} \)  
\textbf{branch-is-path} \ \text{by} \ \text{blast} 
then obtain \( dT1 \ dT2 \ \text{where} \ \text{delete} \ ds \ T = \text{Branching} \ dT1 \ dT2 \ \text{by} \ \text{auto} \)  
then have \( \exists \ T1 \ T2. \ T=\text{Branching} \ T1 \ T2 \)

16
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 treesize-delete: internal \( p \) \( T \) \( \implies \) \( \text{treesize (delete} p \ T) < \text{treesize} T \)

proof (induction \( p \) arbitrary: \( T \))
  case (\text{Nil})
  then have \( \exists \ T1 \ T2. \ T = \text{Branching} \ T1 \ T2 \) by (cases \( T \)) auto
  then obtain \( \ T1 \ T2 \) where \( \text{T1T2-p:} \ T = \text{Branching} \ T1 \ T2 \) by auto
  then show \( ?\text{case by auto} \)
next
  case (\text{Cons} \( a \) \( p \))
  then have \( \exists \ T1 \ T2. \ T = \text{Branching} \ T1 \ T2 \) using path-inv-Cons internal-is-path
  by blast
  then obtain \( \ T1 \ T2 \) where \( \text{T1T2-p:} \ T = \text{Branching} \ T1 \ T2 \) by auto
  show \( ?\text{case by auto} \)

fun cutoff :: (\text{dir list} \Rightarrow \text{bool}) \Rightarrow \text{dir list} \Rightarrow \text{tree} \Rightarrow \text{tree} where
  cutoff \( \text{red} \ \text{ds} \ (\text{Branching} \ T1 \ T2) = \)
  \begin{cases}
    \text{Leaf} & \text{if red} \ \text{ds} \ \text{then Leaf else Branching (cutoff red (ds@[Left]} T1) (cutoff red (ds@[Right]} T2))
    | \text{cutoff red } \text{ds Leaf = Leaf}
  \end{cases}

Initially you should call \( \text{cutoff} \) with \( \text{ds} = [] \). If all branches are red, then \( \text{cutoff} \) gives a subtree. If all branches are red, then so are the ones in \( \text{cutoff} \).
The internal paths of \( \text{cutoff} \) are not red.
then have \( \text{treesize} \left( \text{cutoff red} \ (ds@\text{[Left]}) \ T1 \right) + \text{treesize} \left( \text{cutoff red} \ (ds@\text{[Right]}) \ T2 \right) \leq \text{treesize} \ T1 + \text{treesize} \ T2 \) using add-mono by blast

then show \( \text{?case} \) by auto

qed

abbreviation anypath :: tree \( \Rightarrow \) (dir list \( \Rightarrow \) bool) \( \Rightarrow \) bool where

anypath \( T \ P \) \( \equiv \forall \ p . \ \text{path} \ p \ T \negrightarrow \ P \ p \)

abbreviation anybranch :: tree \( \Rightarrow \) (dir list \( \Rightarrow \) bool) \( \Rightarrow \) bool where

anybranch \( T \ P \) \( \equiv \forall \ p . \ \text{branch} \ p \ T \negrightarrow \ P \ p \)

abbreviation anyinternal :: tree \( \Rightarrow \) (dir list \( \Rightarrow \) bool) \( \Rightarrow \) bool where

anyinternal \( T \ P \) \( \equiv \forall \ p . \ \text{internal} \ p \ T \negrightarrow \ P \ p \)

lemma cutoff-branch’:

anybranch \( T \ (\lambda b . \ \text{red} \ (ds\ @ b)) \) \( \Rightarrow \) anybranch \( \ (\text{cutoff red} \ ds \ T) \ (\lambda b . \ \text{red} \ (ds\ @ b)) \)

proof (induction \( T \) arbitrary: \( ds \))

case (Leaf)

let \( \text{?T} = \ \text{cutoff red} \ ds \ Leaf \)

{ fix \( b \)

assume branch \( b \) \( ?T \)

then have branch \( b \) \( \text{Leaf} \) by auto

then have \( \text{red}(ds@b) \) using \( \text{Leaf} \) by auto
}

then show ?case by simp
	next

case (Branching \( T_1 \ T_2 \))

let \( ?T = \ \text{cutoff red} \ ds \) (Branching \( T_1 \ T_2 \))

from Branching have \( \forall \ p . \ \text{branch} \ (\text{Left} \#p) \ (\text{Branching} \ T_1 \ T_2) \negrightarrow \ \text{red} \ (ds \ @ (\text{Left} \#p)) \) by blast

then have \( \forall \ p . \ \text{branch} \ p \ T_1 \negrightarrow \ \text{red} \ (ds \ @ (\text{Left} \#p)) \) by auto

then have \( \text{anybranch} \ T_1 \ (\lambda p . \ \text{red} \ ((ds @ [\text{Left}]) @ p)) \) by auto

then have \( \text{aa}: \ \text{anybranch} \ (\text{cutoff red} \ (ds \ @ \text{[Left]}) \ T_1) \ (\lambda p . \ \text{red} \ ((ds @ [\text{Left}]) @ p)) \)

using Branching by blast

from Branching have \( \forall \ p . \ \text{branch} \ (\text{Right} \#p) \ (\text{Branching} \ T_1 \ T_2) \negrightarrow \ \text{red} \ (ds \ @ (\text{Right} \#p)) \) by blast

then have \( \forall \ p . \ \text{branch} \ p \ T_2 \negrightarrow \ \text{red} \ (ds \ @ (\text{Right} \#p)) \) by auto

then have \( \text{anybranch} \ T_2 \ (\lambda p . \ \text{red} \ ((ds @ [\text{Right}]) @ p)) \) by auto

then have \( \text{bb}: \ \text{anybranch} \ (\text{cutoff red} \ (ds \ @ \text{[Right]}) \ T_2) \ (\lambda p . \ \text{red} \ ((ds @ [\text{Right}]) @ p)) \)

using Branching by blast

{ fix \( b \)

assume \( b-p: \ \text{branch} \ b \ ?T \)

have \( \text{red} \ ds \ \lor \ \neg \text{red} \ ds \) by auto

then have \( \text{red}(ds@b) \)
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' ∧ (a → branch b' ?T_1') ∧
  (¬a → branch b' ?T_2') using branch-inv-Branching[a 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) → anybranch (cutoff red []) T
(λp. red p)
  using cutoff-branch[of T red []] by auto

lemma cutoff-internal':
  anybranch T (λb. red(ds@b)) → 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 ∀p. branch (Left#p) (Branching T_1 T_2) → red (ds @ (Left#p)) by blast
  then have ∀p. branch p T_1 → red (ds @ (Left#p)) by auto
  then have anybranch T_1 (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 ∀p. branch (Right#p) (Branching T_1 T_2) → red (ds @ (Right#p)) by blast
  then have ∀p. branch p T_2 → red (ds @ (Right#p)) by auto
  then have anybranch T_2 (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=[] ∨ p=#] by auto
  then have ¬red(ds@p)
  proof
assume $p = []$ then show $\neg \text{red}(ds @ p)$ using $ds-p$ by auto

next
let $?T_1' = \text{cutoff red} (ds@[Left]) T_1$
let $?T_2' = \text{cutoff red} (ds@[Right]) T_2$
assume $p \neq []$
moreover have $?T = \text{Branching} ?T_1' ?T_2'$ using $ds-p$ by auto
ultimately obtain $a p'$ where $b-p$: $p = a \# p' \land$
\begin{align*}
(a \rightarrow \text{internal} p' (\text{cutoff red} (ds@[Left]) T_1)) \land \\
(\neg a \rightarrow \text{internal} p' (\text{cutoff red} (ds@[Right]) T_2))
\end{align*}
using $b-p$ internal-inv-Branching[of $p$ $?T_1'$ $?T_2'$] by auto
then have $\neg \text{red}(ds@[a] @ p')$ using $aa bb$ by (cases a) auto
then show $\neg \text{red}(ds@[p])$ using $b-p$ by simp

qed

} then show $?\text{case}$ by blast

qed

lemma cutoff-internal: anybranch $T$ red $\Rightarrow$ anyinternal (cutoff red $[]$ $T$) ($\lambda p. \neg \text{red} p$)
using cutoff-internal[of $T$ red $[]$] by auto

lemma cutoff-branch-internal':
anybranch $T$ red $\Rightarrow$ anyinternal (cutoff red $[]$ $T$) ($\lambda p. \neg \text{red} p$) $\land$ anybranch (cutoff red $[]$ $T$) ($\lambda p. \text{red} p$)
using cutoff-internal[of $T$] cutoff-branch[of $T$] by blast

lemma cutoff-branch-internal:
anybranch $T$ red $\Rightarrow$ $\exists T'$. anyinternal $T'$ ($\lambda p. \neg \text{red} p$) $\land$ anybranch $T'$ ($\lambda p. \text{red} p$
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 $\Rightarrow$ bool where
$\text{wf-tree} T \equiv (\forall ds d. (ds @ d) \in T \rightarrow ds \in T)$

The subtree in with root $r$

fun subtree :: dir list set $\Rightarrow$ dir list $\Rightarrow$ dir list set where
$\text{subtree} T r = \{ds \in T. \exists ds'. ds = r \mapsto ds'\}$

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

lemma subtree-pos:
$\text{subtree} T ds \subseteq \text{subtree} T (ds@[Left]) \cup \text{subtree} T (ds@[Right]) \cup \{ds\}$
proof (rule subsetI; rule Set.UnCI)
  let ?subtree = subtree T
  fix x
  assume asm: x ∈ ?subtree ds
  assume x /∈ ds
  then have x ≠ ds by simp
  then have ∃ e d. x = ds @ [d] @ e using asm.exhaust by auto
  then have (∃ e. x = ds @ [Left] @ e) ∨ (∃ e. x = ds @ [Right] @ e) using bool.exhaust by auto
  then show x ∈ ?subtree (ds @ [Left]) ∪ ?subtree (ds @ [Right]) using asm by auto
qed

3.1 Infinite Paths
abbreviation wf-infpath :: (nat ⇒ 'a list) ⇒ bool where
  wf-infpath f ≡ (f 0 = []) ∧ (∀ n. ∃ a. f (Suc n) = (f n) @ [a])
lemma infpath-length: wf-infpath f ⇒ 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 ⇒ n1 ≤ n2 ⇒ ∃ a. (f n1) @ a = (f n2)
  proof (induction n2)
    case (Suc n2)
    then have n1 ≤ n2 ∨ n1 = Suc n2 by auto
    then show ?case
      proof
        assume n1 ≤ n2
        then obtain a where a: f n1 @ a = f n2 using Suc by auto
        have b: ∃ b. f (Suc n2) = f n2 @ [b] using Suc by auto
        from a b have ∃ b. f n1 @ (a @ [b]) = f (Suc n2) by auto
        then show ∃ c. f n1 @ c = f (Suc n2) by blast
      next
      assume n1 = Suc n2
      then have f n1 @ [] = f (Suc n2) by auto
      then show ∃ a. f n1 @ a = f (Suc n2) 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 n1)
  assumes n1n2: n1 ≤ n2
  ...
shows \( f_{n_1} \uparrow i = f_{n_2} \uparrow i \)

**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 \( a: f_{n_1} @ a = f_{n_2} \) by auto
have \( (f_{n_1} @ a) \uparrow i = f_{n_1} \uparrow i \) using smalli by (simp add: nth-append)
then show \( \text{thesis} \) using \( a: \) by auto

**qed**

4 König’s Lemma

**lemma inf-subs:**

assumes \( \text{inf} : \neg \text{finite}(\text{subtree } T \text{ ds}) \)
shows \( \neg \text{finite}(\text{subtree } T (\text{ds} @ [\text{\text{Left}}])) \lor \neg \text{finite}(\text{subtree } T (\text{ds} @ [\text{\text{Right}}])) \)

**proof**

let \( \text{?subtree} = \text{subtree } T \)

\begin{align*}
\text{assume asms: finite(?subtree(ds @ [\text{\text{Left}}]))} \\
\text{finite(?subtree(ds @ [\text{\text{Right}}]))} \\
\text{have ?subtree ds } &\subseteq \text{ ?subtree (ds @ [\text{\text{Left}}] ) \lor ?subtree (ds @ [\text{\text{Right}}]) \lor \{ds\}} \\
\text{using subtree-pos by auto} \\
\text{then have finite(?subtree (ds)) using asms by (simp add: finite-subset)} \\
\end{align*}

then show \( \neg \text{finite(?subtree (ds))} \) using \( \text{inf by auto} \)

**qed**

fun buildchain :: (dir list ⇒ dir list) ⇒ nat ⇒ dir list where

\[ \text{buildchain next 0} = [] \]
\[ \text{buildchain next } (\text{Suc } n) = \text{next } (\text{buildchain next } n) \]

**lemma konig:**

assumes \( \text{inf} : \neg \text{finite } T \)
assumes wellformed: \( \text{wf-tree } T \)
shows \( \exists c. \text{wf-infpath } c \land (\forall n. (c n) \in T) \)

**proof**

let \( \text{?subtree} = \text{subtree } T \)
let \( \text{?nextnode} = \lambda ds. (\text{if } \neg \text{finite (?subtree (ds @ [\text{\text{Left}}])) then ds @ [\text{\text{Left}}] else ds @ [\text{\text{Right}}]} \)

\[ \text{let } ?c = \text{buildchain } ?\text{nextnode} \]

have is-chain: \( \text{wf-infpath } ?c \) by auto

from wellformed have prefix: \( \land \text{ds d. (ds @ d) }\in T \implies ds \in T \) by blast

\begin{align*}
\text{fix } n \\
\text{have } (?c n) \in T \land \neg \text{finite (?subtree (?c n))} \\
\text{proof } (\text{induction } n) \\
\end{align*}

24
case 0
  have \( \exists ds. \, ds \in T \) using \( \inf \) by (simp add: not-finite-existsD)
  then obtain \( ds \) where \( ds \in T \) by auto
  then have \( [] @ ds \in T \) by auto
  then obtain \( ds \) where \( ds \in T \) by auto
  then have \( [] \in T \) using prefix[of []] by auto
  then show \( \text{?case using } \inf \) by auto
next
  case \( \text{(Suc } n) \)
  from Suc have next-in: \( (?c \, n) \in T \) by auto
  from Suc have next-inf: \( \neg\text{finite } (?\text{subtree } (?c \, n)) \) by auto
  from next-inf have next-next-inf:
    \( \neg\text{finite } (?\text{subtree } (?\text{nextnode } (?c \, n))) \)
    using inf-subs by auto
  then obtain \( ds \) where \( ds \in ?\text{subtree } (?\text{nextnode } (?c \, n)) \) by auto
  then have \( ds \in T \) using prefix[of ?nextnode (?c n) suf] by auto
  then have \( (?c \, \text{(Suc } n)) \in T \) by auto
  then show \( \text{?case using } \text{next-next-inf} \) by auto
qed

\}
then show \( \text{wf-infpath } ?c \land (\forall n. \, (?c \, n) \in T) \) using \( \text{is-chain} \) by auto
qed

end

5 More Terms and Literals

theory Resolution imports TermsAndLiterals Tree begin

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

lemma cancel-comp1: \( (l^c)^c = l \) by (cases \( l \)) auto

lemma cancel-comp2:
  assumes \( \text{asm}: \, l_1^c = l_2^c \)
  shows \( l_1 = l_2 \)
proof -
  from \( \text{asm} \) have \( (l_1^c)^c = (l_2^c)^c \) by auto
  then have \( l_1 = (l_2)^c \) using cancel-comp1[of \( l_1 \)] by auto
  then show \( \text{?thesis using } \text{cancel-comp1[of } l_2 \) by auto
qed

25
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 \( \text{thesis} \) by auto
qed

lemma sign-comp: \( \text{sign } l_1 \neq \text{sign } l_2 \land \text{get-pred } l_1 = \text{get-pred } l_2 \land \text{get-terms } l_1 = \text{get-terms } l_2 \longleftrightarrow l_1 = l_2^c \)
by (cases l_1; cases l_2) auto

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

6 Clauses

type-synonym \('t\ clause = 't literal set\)

abbreviation complementls :: \('t\ literal set \Rightarrow 't\ literal set\) \(\{L^C \equiv \text{complement } L\}\) where
\(L^C \equiv \text{complement } \cdot L\)

lemma cancel-compls1: \((L^C)^C = L\)
apply (auto simp add: cancel-comp1)
apply (metis imageI cancel-comp1)
done

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

fun vars_t :: \('\term\ \Rightarrow \var-sym\ set\) where
\( \text{vars}_t \ (\text{Var } x) = \{x\} \)
| \text{vars}_t (\text{Fun } f \ ts) = (\bigcup t \in \text{set } ts. \text{vars}_t t) \)
abbreviation \( \text{vars}_{ts} :: \text{fterm list} \Rightarrow \text{var-sym set} \) where
\[
\text{vars}_{ts} \ ts \equiv (\bigcup t \in \text{set} \ ts. \ \text{vars} \ t)
\]
definition \( \text{vars}_l :: \text{fterm literal} \Rightarrow \text{var-sym set} \) where
\[
\text{vars}_l \ l = \text{vars}_{ts} (\text{get-terms} \ l)
\]
definition \( \text{vars}_{ls} :: \text{fterm literal set} \Rightarrow \text{var-sym set} \) where
\[
\text{vars}_{ls} \ L \equiv \bigcup \ l \in L. \ \text{vars}_l \ l
\]

lemma ground-vars: \( \text{ground}_t \ t \Rightarrow \text{vars}_t \ t = \{\} \)
by (induction \( t \)) auto

lemma ground-vars-ts: \( \text{ground}_{ts} \ ts \Rightarrow \text{vars}_{ts} \ ts = \{\} \)
using ground-vars \( \text{by} \) auto

lemma ground-vars-ls: \( \text{ground}_{ls} \ L \Rightarrow \text{vars}_{ls} \ L = \{\} \)
unfolding \( \text{vars}_{ls} \)-def using ground-vars-ts \( \text{by} \) auto

lemma ground-comp: \( \text{ground}_l (l^c) \leftarrow\rightarrow \text{ground}_l \ l \) by (cases \( l \)) auto

lemma ground-comp-ls: \( \text{ground}_{ls} (L^c) \leftarrow\rightarrow \text{ground}_{ls} \ L \) using ground-comp \( \text{by} \) auto

7 Semantics

type-synonym \( 'u \text{ fun-denot} = \text{fun-sym} \Rightarrow 'u \ list \Rightarrow 'u \)
type-synonym \( 'u \text{ pred-denot} = \text{pred-sym} \Rightarrow 'u \ list \Rightarrow \text{bool} \)
type-synonym \( 'u \text{ var-denot} = \text{var-sym} \Rightarrow 'u \)

fun eval\( _t :: 'u \text{ var-denot} \Rightarrow 'u \text{ fun-denot} \Rightarrow \text{fterm} \Rightarrow 'u \)
where
\[
\text{eval}_t \ E \ F \ (\text{Var} \ x) = E \ x
\]
\[
\mid \text{eval}_t \ E \ F \ (\text{Fun} \ f \ ts) = F \ f \ (\text{map} \ (\text{eval}_t \ E \ F) \ ts)
\]

abbreviation eval\( _{ts} :: 'u \text{ var-denot} \Rightarrow 'u \text{ fun-denot} \Rightarrow \text{fterm list} \Rightarrow 'u \ list \)
where
\[
\text{eval}_{ts} \ E \ F \ ts \equiv \text{map} \ (\text{eval}_t \ E \ F) \ ts
\]

fun eval\( _l :: 'u \text{ var-denot} \Rightarrow 'u \text{ fun-denot} \Rightarrow 'u \text{ pred-denot} \Rightarrow \text{fterm literal} \Rightarrow \text{bool} \)
where
\[
\text{eval}_l \ E \ F \ G \ (\text{Pos} \ p \ ts) \leftarrow G \ p \ (\text{eval}_{ts} \ E \ F \ ts)
\]
\[
\mid \text{eval}_l \ E \ F \ G \ (\text{Neg} \ p \ ts) \leftarrow \neg G \ p \ (\text{eval}_{ts} \ E \ F \ ts)
\]
definition eval\( _c :: 'u \text{ fun-denot} \Rightarrow 'u \text{ pred-denot} \Rightarrow \text{fterm clause} \Rightarrow \text{bool} \)
where
\[
\text{eval}_c \ F \ G \ C \leftarrow (\forall E. \exists l \in C. \ \text{eval}_l \ E \ F \ G \ l)
\]
definition eval\( _{cs} :: 'u \text{ fun-denot} \Rightarrow 'u \text{ pred-denot} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool} \)
where
$$\text{eval}_{c,s} \ F \ G \ Cs \leftrightarrow (\forall C \in Cs. \text{eval}_{c} \ F \ G \ C)$$

### 7.1 Semantics of Ground Terms

**Lemma** ground-var-denott: \( \text{ground}_{t} t \implies (\text{eval}_{t} E \ F \ t = \text{eval}_{t} E' \ F \ t) \)

**Proof** (induction \( t \))

- **Case** \((\text{Var} \ x)\)
  - then have \(\text{False} \) by \text{auto}
  - then show \(\text{?case} \) by \text{auto}

- **Case** \((\text{Fun} \ f \ ts)\)
  - then have \((\forall t \in \text{set} \ ts. \text{ground}_{t} t) \) by \text{auto}
  - then have \((\forall t \in \text{set} \ ts. \text{eval}_{t} E \ F \ t = \text{eval}_{t} E' \ F \ t \) using \text{Fun} by \text{auto}
  - then have \(F \ f \ (\text{map} (\text{eval}_{t} E \ F) \ ts) = F \ f \ (\text{map} (\text{eval}_{t} E' \ F) \ ts) \) by \text{metis}
  - then show \(\text{?case} \) by \text{simp}

**QED**

**Lemma** ground-var-denotts: \( \text{ground}_{ts} ts \implies (\text{eval}_{ts} E \ F \ ts = \text{eval}_{ts} E' \ F \ ts) \)

**Using** ground-var-denott by (metis \text{map-eq-conv})

**Lemma** ground-var-denot: \( \text{ground}_{l} l \implies (\text{eval}_{l} E \ F \ G \ l = \text{eval}_{l} E' \ F \ G \ l) \)

**Proof** (induction \( l \))

- **Case** \(\text{Pos} \) then show \(\text{?case} \) using ground-var-denotts by (metis eval.l.simps(1) literal.sel(3))

- **Next**
  - **Case** \(\text{Neg} \) then show \(\text{?case} \) 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 \( \cdot \_ \ 55 \)) where

- \((\text{Var} \ x) \cdot \_ \_ \sigma = \sigma \ x \)
- \((\text{Fun} \ f \ ts) \cdot \_ \_ \sigma = F \ f \ (\text{map} (\lambda t. \cdot \_ \_ \sigma) \ ts) \)

**Abbreviation** subs :: fterm list ⇒ substitution ⇒ fterm list (infixl \( \cdot \_ \_ \ 55 \)) where

- \(ts \cdot \_ \_ \sigma \equiv (\text{map} (\lambda t. \cdot \_ \_ \sigma) \ ts) \)

**Fun** subl :: fterm literal ⇒ substitution ⇒ fterm literal (infixl \( \cdot \_ \ 55 \)) where

- \((\text{Pos} \ p \ ts) \cdot \_ \_ \sigma = \text{Pos} \ p \ (ts \cdot \_ \_ \sigma) \)
- \((\text{Neg} \ p \ ts) \cdot \_ \_ \sigma = \text{Neg} \ p \ (ts \cdot \_ \_ \sigma) \)

**Abbreviation** subls :: fterm literal set ⇒ substitution ⇒ fterm literal set (infixl \( \cdot \_ \ 55 \)) where

- \(L \cdot \_ \_ \sigma \equiv (\lambda l. \cdot \_ \_ \sigma) \ l \)

28
lemma subls-def2: $L \cdot l_s \sigma = \{ l \cdot l \sigma | l. \ l \in L \}$ by auto

definition instance-of-t :: $\text{fterm} \Rightarrow \text{fterm} \Rightarrow \text{bool}$ where
instance-of_t $t_1 \ t_2 \leftarrow (\exists \sigma. \ t_1 = t_2 \cdot l \sigma)$

definition instance-of-ts :: $\text{fterm list} \Rightarrow \text{fterm list} \Rightarrow \text{bool}$ where
instance-of_ts $t_{s_1} \ t_{s_2} \leftarrow (\exists \sigma. \ t_{s_1} = t_{s_2} \cdot l_s \sigma)$

definition instance-of_l :: $\text{fterm literal} \Rightarrow \text{fterm literal} \Rightarrow \text{bool}$ where
instance-of_l $l_1 \ l_2 \leftarrow (\exists \sigma. \ l_1 = l_2 \cdot l \sigma)$

definition instance-of_ls :: $\text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{bool}$ where
instance-of_ls $C_1 \ C_2 \leftarrow (\exists \sigma. \ C_1 = C_2 \cdot l_s \sigma)$

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

lemma compls-subls: $(L^C) \cdot l_s \sigma = (L \cdot l_s \sigma)^C$
using comp-sub apply auto
apply (metis image-eqI)
done

lemma subls-union: $(L_1 \cup L_2) \cdot l_s \sigma = (L_1 \cdot l_s \sigma) \cup (L_2 \cdot l_s \sigma)$ by auto

8.1 The Empty Substitution

abbreviation $\varepsilon :: \text{substitution}$ where
$\varepsilon \equiv \text{Var}$

lemma empty-subt: $(t :: \text{fterm}) \cdot t \ \varepsilon = t$
by (induction $t$) (auto simp add: map-idI)

lemma empty-subts: $ts \cdot t_s \ \varepsilon = ts$
using empty-subt by auto

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

lemma empty-subls: $L \cdot l_s \ \varepsilon = L$
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
unfolding instance-of-t-def
proof
  show $ts = ts \cdot ts \varepsilon$ using empty-subts by auto
qed

lemma instance-of-l-self: instance-of 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 ls
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 $t \mapsto t \cdot t \sigma = t$
by (induction $t$) (auto simp add: map-idI)

lemma ground-subst: ground $ts \mapsto ts \cdot ts \sigma = ts$
using ground-sub by (simp add: map-idI)

lemma ground1-subst: ground $l \mapsto l \cdot l \sigma = l$
using ground-subst by (cases $l$) auto

lemma ground1s-substs:
  assumes ground: ground $l$ $s$
  shows $L \cdot l \sigma = L$
proof
  {
    fix $l$
    assume $L: l \in L$
    then have $l = l \cdot l \sigma$ using $l$ ground by auto
    moreover then have $l \cdot l \sigma = L \cdot l \sigma$ using $l$ ground1-subst by auto
    ultimately have $l \in L \cdot l \sigma$ by auto
  }
moreover
  {
    fix $l$
  }

30
assume \( l \in L \vdash_s \sigma \)
then obtain \( l' \) where \( l' \in L \land l' \vdash \sigma = l \) by auto
then have \( l' = l \) using ground ground\_l-subs by auto
from \( l \in L \vdash \) this have \( l \in L \) by auto
}
ultimately show \( \text{thesis} \) by auto
qed

8.3 Composition

definition composition :: substitution \( \Rightarrow \) substitution \( \Rightarrow \) substitution \((\text{infixl} \cdot 55)\)
where
\((\sigma_1 \cdot \sigma_2) \, x = (\sigma_1 \, x) \, \vdash \sigma_2\)

lemma composition\_conseq2t: \((t \, \vdash \sigma_1) \, \vdash \sigma_2 = t \, \vdash (\sigma_1 \cdot \sigma_2)\)
proof (induction t)
  case (Var x)
  have \(((\text{Var} \, x) \, \vdash \sigma_1) \, \vdash \sigma_2 = (\sigma_1 \, x) \, \vdash \sigma_2\) by simp
  also have \(... = (\sigma_1 \cdot \sigma_2) \, x\) unfolding composition\_def by simp
  finally show \(\text{case} \) by auto
next
  case (Fun t ts)
  then show \(\text{case}\) unfolding composition\_def by auto
qed

lemma composition\_conseq2ts: \((ts \, \vdash \sigma_1) \, \vdash \sigma_2 = ts \, \vdash (\sigma_1 \cdot \sigma_2)\)
  using composition\_conseq2t by auto

lemma composition\_conseq2l: \((l \, \vdash \sigma_1) \, \vdash \sigma_2 = l \, \vdash (\sigma_1 \cdot \sigma_2)\)
  using composition\_conseq2t by (cases l) auto

lemma composition\_conseq2ls: \((l \, \vdash s, \sigma_1) \, \vdash \sigma_2 = l \, \vdash s (\sigma_1 \cdot \sigma_2)\)
  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-of1-trans :
  assumes \(t_{12}:\) instance-of\(_1\) \(t_1 t_2\)
  assumes \(t_{23}:\) instance-of\(_1\) \(t_2 t_3\)
  shows instance-of\(_1\) \(t_1 t_3\)
proof
  from \(t_{12}\) obtain \(\sigma_{12}\) where \(t_1 = t_2 \cdot \sigma_{12}\)
  unfolding instance-of\(_1\)-def by auto
moreover
  from \(t_{23}\) obtain \(\sigma_{23}\) where \(t_2 = t_3 \cdot \sigma_{23}\)
  unfolding instance-of\(_1\)-def by auto
ultimately
  have \(t_1 = (t_3 \cdot \sigma_{23}) \cdot \sigma_{12}\) by auto
  then have \(t_1 = t_3 \cdot (\sigma_{23} \cdot \sigma_{12})\) using composition-conseq2t by simp
  then show \(\text{?thesis}\) unfolding instance-of\(_1\)-def by auto
qed

lemma instance-of1s-trans :
  assumes \(ts_{12}:\) instance-of\(_{1s}\) \(ts_1 ts_2\)
  assumes \(ts_{23}:\) instance-of\(_{1s}\) \(ts_2 ts_3\)
  shows instance-of\(_{1s}\) \(ts_1 ts_3\)
proof
  from \(ts_{12}\) obtain \(\sigma_{12}\) where \(ts_1 = ts_2 \cdot \sigma_{12}\)
  unfolding instance-of\(_{1s}\)-def by auto
moreover
  from \(ts_{23}\) obtain \(\sigma_{23}\) where \(ts_2 = ts_3 \cdot \sigma_{23}\)
  unfolding instance-of\(_{1s}\)-def by auto
ultimately
  have \(ts_1 = (ts_3 \cdot \sigma_{23}) \cdot \sigma_{12}\) by auto
  then have \(ts_1 = ts_3 \cdot (\sigma_{23} \cdot \sigma_{12})\) using composition-conseq2ts by simp
  then show \(\text{?thesis}\) unfolding instance-of\(_{1s}\)-def by auto
qed

lemma instance-of1-trans :
  assumes \(l_{12}:\) instance-of\(_1\) \(l_1 l_2\)
  assumes \(l_{23}:\) instance-of\(_1\) \(l_2 l_3\)
  shows instance-of\(_1\) \(l_1 l_3\)
proof
  from \(l_{12}\) obtain \(\sigma_{12}\) where \(l_1 = l_2 \cdot \sigma_{12}\)
  unfolding instance-of\(_1\)-def by auto
moreover
  from \(l_{23}\) obtain \(\sigma_{23}\) where \(l_2 = l_3 \cdot \sigma_{23}\)
  unfolding instance-of\(_1\)-def by auto
ultimately
have \( l_1 = (l_3 \cdot \sigma_{23} \cdot \sigma_{12}) \) by auto
then have \( l_1 = l_3 \cdot (\sigma_{23} \cdot \sigma_{12}) \) using composition-conseq2l by simp
then show \(?thesis\) unfolding instance-of-def by auto
qed

8.4 Merging substitutions

lemma project-sub:
  assumes \( \text{inst-C}: C \vdash l \ lmbd = C' \)
  assumes \( \text{L' \subseteq C'} \)
  shows \( \exists \ L \subseteq C. \ L \vdash l \ lmbd = L' \land (C - L) \vdash l \ lmbd = C' - L' \)
proof –
  let \(?L = \{l \in C. \ \exists l' \in L'. \ l \ lmbd = l'\} \)
  have \( \exists L \subseteq C \) by auto
  moreover
  have \( \forall l \ lmbd = L' \)
  proof (rule Orderings.order-antisym; rule Set.subsetI)
    fix \( l' \)
    assume \( l' \in L' \)
    from \( \text{inst-C} \) have \( \{l \ lmbd |. l \in C\} = C' \) unfolding subls-def2 by –
    then have \( \exists l. \ l' = l \ lmbd \land l \in C \land l \ lmbd \in L' \) using \( L\sub' \ l' \) by auto
    then have \( l' \in \{l \in C. \ l \ lmbd \in L'\} \ l_{\ lmbd} = l' \) by auto
    then show \( l' \in \{l \in C. \ \exists l' \in L'. \ l \ lmbd = l'\} \ l_{\ lmbd} = l' \) by auto
    qed auto
  moreover
  have \( (C - ?L) \ l_{\ lmbd} = C' - L' \) using \( \text{inst-C} \) by auto
  moreover
  ultimately show \(?thesis\) by auto
  qed

lemma relevant-vars-subt:
∀ x ∈ vars, t, σ₁ x = σ₂ x \implies t \cdot_1 σ₁ = t \cdot_1 σ₂

proof (induction t)
  case (Fun f ts)
  have f: ∀ t. t ∈ set ts \implies vars⁺ t ⊆ vars⁺ ts by (induction ts) auto
  have ∀ t ∈ set ts. t \cdot_1 σ₁ = t \cdot_1 σ₂
    proof
    fix t
    assume tints: t ∈ set ts
    then have ∀ x ∈ vars⁺ t. σ₁ x = σ₂ x using f Fun(2) by auto
    then show t \cdot_1 σ₁ = t \cdot_1 σ₂ using Fun tints by auto
  qed
  then have ts \cdot_1 σ₁ = ts \cdot_1 σ₂ by auto
  then show ?thesis by auto
qed auto

lemma relevant-vars-subts:
assumes asm: ∀ x ∈ vars⁺ ts. σ₁ x = σ₂ x
shows ts \cdot_1 σ₁ = ts \cdot_1 σ₂
proof
  have f: ∀ t. t ∈ set ts \implies vars⁺ t ⊆ vars⁺ ts by (induction ts) auto
  have ∀ t ∈ set ts. t \cdot_1 σ₁ = t \cdot_1 σ₂
    proof
    fix t
    assume tints: t ∈ set ts
    then have ∀ x ∈ vars⁺ t. σ₁ x = σ₂ x using f asm by auto
    then show t \cdot_1 σ₁ = t \cdot_1 σ₂ using relevant-vars-subt tints by auto
  qed
  then show ?thesis by auto
qed auto

lemma relevant-vars-subl:
∀ x ∈ vars⁺ l. σ₁ x = σ₂ x \implies l \cdot_1 σ₁ = l \cdot_1 σ₂
proof (induction l)
  case (Pos p ts)
  then show ?case using relevant-vars-subts unfolding vars⁺-def by auto
next
  case (Neg p ts)
  then show ?case using relevant-vars-subts unfolding vars⁺-def by auto
qed

lemma relevant-vars-subls:
assumes asm: ∀ x ∈ vars⁺ L. σ₁ x = σ₂ x
shows L \cdot_1 σ₁ = L \cdot_1 σ₂
proof
  have f: \forall l. l ∈ L \implies vars⁺ l ⊆ vars⁺ L unfolding vars⁺-def by auto
  have ∀ l ∈ L. l \cdot_1 σ₁ = l \cdot_1 σ₂
    proof
    fix l
    assume linls: l ∈ L
    qed
then have ∀x∈vars\ l. σ_1 x = σ_2 x using f asm by auto
then show l · σ_1 = l · σ_2 using relevant-vars-subl linls by auto
qed
then show ?thesis by (meson image-cong)
qed

lemma merge-sub:
assumes dist: vars\ l\ C ∩ vars\ l\ D = {}
assumes CC':  C · lmbd = C'
assumes DD':  D · µ = D'
shows ∃η. C · lmbd = C' ∧ D · µ = D'
proof −
let ?η = λx. if x ∈ vars\ l\ C then lmbd x else µ x
have ∀x∈vars\ l\ C. ?η x = lmbd x by auto
then have C · lmbd using relevant-vars-subls\ of C ?η lmbd by auto
then have C · lmbd using CC' by auto
moreover
have ∀x ∈ vars\ l\ D. ?η x = µ x using dist by auto
then have D · µ using relevant-vars-subls\ of D ?η µ by auto
then have D · µ 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 ≡ C · (λx. Var ("1" @ x))

abbreviation std\ 2 :: fterm clause ⇒ fterm clause where
std\ 2 C ≡ C · (λx. Var ("2" @ x))

lemma std-apart-apart'\ :
x ∈ vars\ l\ (t · (λx::char list. Var (y @ x))) ⇒ ∃x'. x = y@x'
by (induction t) auto

lemma std-apart-apart'\ : x ∈ vars\ l\ (t · (λx. Var (y@x))) ⇒ ∃x'. x = y@x'
unfolding vars\ l-def using std-apart-apart'\ by (cases l) auto

lemma std-apart-apart\ : vars\ l\ (std\ 1 C_1) ∩ vars\ l\ (std\ 2 C_2) = {}
proof −
{ fix x
assume zin: x ∈ vars\ l\ (std\ 1 C_1) ∩ vars\ l\ (std\ 2 C_2)
from zin have x ∈ vars\ l\ (std\ 1 C_1) by auto
then have ∃x'. x="1" @ x'
    using std-apart-apart\ of x - "1" unfolding vars\ l-def by auto
moreover
from \( x \in \text{vars} \) (std2 C2) 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 \( ?\text{thesis} \) by auto
qed

lemma std-apart-instance-of\_ls1: instance-of\_ls \( C_1 \) (std\_ls C1)
proof --
  have empty: \( (\lambda x. \text{Var} ("1" @x)) \cdot (\lambda x. \text{Var} (tl x)) = \varepsilon \) using composition-def
by auto

  have \( C_1 \cdot_{\text{ls}} \varepsilon = C_1 \) using empty-subls by auto
then have \( C_1 \cdot_{\text{ls}} ((\lambda x. \text{Var} ("1" @x)) \cdot (\lambda x. \text{Var} (tl x))) = C_1 \) using empty by auto
then have \( (C_1 \cdot_{\text{ls}} (\lambda x. \text{Var} ("1" @x))) \cdot_{\text{ls}} (\lambda x. \text{Var} (tl x)) = C_1 \) using composition-conseq2ls
by auto
then have \( C_1 = (\text{std}_1 C_1) \cdot_{\text{ls}} (\lambda x. \text{Var} (tl x)) \) by auto
then show instance-of\_ls \( C_1 \) (std\_ls C1) unfolding instance-of\_ls-def by auto
qed

lemma std-apart-instance-of\_ls2: instance-of\_ls \( C_2 \) (std\_ls C2)
proof --
  have empty: \( (\lambda x. \text{Var} ("2" @x)) \cdot (\lambda x. \text{Var} (tl x)) = \varepsilon \) using composition-def
by auto

  have \( C_2 \cdot_{\text{ls}} \varepsilon = C_2 \) using empty-subls by auto
then have \( C_2 \cdot_{\text{ls}} ((\lambda x. \text{Var} ("2" @x)) \cdot (\lambda x. \text{Var} (tl x))) = C_2 \) using empty
by auto
then have \( (C_2 \cdot_{\text{ls}} (\lambda x. \text{Var} ("2" @x))) \cdot_{\text{ls}} (\lambda x. \text{Var} (tl x)) = C_2 \) using composition-conseq2ls
by auto
then have \( C_2 = (\text{std}_2 C_2) \cdot_{\text{ls}} (\lambda x. \text{Var} (tl x)) \) by auto
then show instance-of\_ls \( C_2 \) (std\_ls C2) unfolding instance-of\_ls-def by auto
qed

9 Unifiers

definition unifier\_ls :: substitution \( \Rightarrow \) fterm set \( \Rightarrow \) bool where
unifier\_ls \( \sigma \) \( ts \) \( \leftarrow \) (\( \exists t'. \forall t \in ts. t \cdot_{\text{t}} \sigma = t' \))
definition unifier\_ls :: substitution \( \Rightarrow \) fterm literal set \( \Rightarrow \) bool where
unifier\_ls \( \sigma \) \( L \) \( \leftarrow \) (\( \exists l'. \forall l \in L. l \cdot_{\text{l}} \sigma = l' \))

lemma unif-sub:
  assumes unif: unifier\_ls \( \sigma \) \( L \)
  assumes nonempty: \( L \neq \{ \} \)

36
shows \( \exists l. \text{subls } L \sigma = \{ \text{subl } l \sigma \} \)

proof –

from nonempty obtain \( l \) where \( l \in L \) by auto
from unif this have \( L \cdot\sigma = \{ l \cdot\sigma \} \) unfolding unifier\( L \sigma \)-def by auto
then show \(?thesis\) by auto
qed

lemma unifier\( L \sigma \)-def2:

assumes \( L\text{-elem}: L \neq \{ \} \)
shows \( \text{unifier}\( L \sigma \) \) \( L \leftrightarrow (\exists l. (\lambda t. \text{sub } t \sigma) \cdot ts = \{ l \}) \)

proof

assume unif: \( \text{unifier}\( L \sigma \) \) \( L \)
from \( L\text{-elem} \) obtain \( t \) where \( t \in ts \) by auto
then have \((\lambda t. \text{sub } t \sigma) \cdot ts = \{ t \cdot \sigma \} \) using unif unfolding unifier\( L \sigma \)-def by auto
then show \( \exists l. (\lambda t. \text{sub } t \sigma) \cdot ts = \{ l \} \) by auto
next

assume \( \exists l. (\lambda t. \text{sub } t \sigma) \cdot ts = \{ l \} \)
then obtain \( l \) where \((\lambda t. \text{sub } t \sigma) \cdot ts = \{ l \} \) by auto
then have \( \forall l' \in ts. l' \cdot \sigma = l \) by auto
then show \( \text{unifier}\( L \sigma \) \) \( ts \) unfolding unifier\( L \sigma \)-def by auto

qed

lemma  \( \text{ground}\( L \sigma \)-unif-singleton:

assumes ground\( L \sigma \): ground\( L \sigma \) \( L \)
assumes unif: \( \text{unifier}\( L \sigma \) \) \( L \sigma \) \( \sigma \)
assumes empt: \( L \neq \{ \} \)
shows \( \exists l. L = \{ l \} \)

proof –

from unif empt have \( \exists l. L \cdot\sigma' = \{ l \} \) using unif-sub by auto
then show \(?thesis\) using ground\( L \sigma \)-subls ground\( L \sigma \) by auto

qed

definition \( \text{unifiablets} :: \text{fterm set} \Rightarrow \text{bool} \) where
unifiablets $fs \iff (\exists \sigma. \unifier_{ts} \sigma fs)$

**Definition**

unifiablets $::=$ fterm literal set $\Rightarrow$ bool

where

unifiablets $L \iff (\exists \sigma. \unifier_{ts} \sigma L)$

**Lemma**

unifier-comp[simp]: $\unifier_{ts} \sigma (L^C) \iff \unifier_{ts} \sigma L$

**Proof**

assume $\unifier_{ts} \sigma (L^C)$

then obtain $l''$ where $l''$-p: $\forall l \in L^C. l \cdot \sigma = l''$

unfolding $\unifier_{ts}$-def by auto

obtain $l'$ where $(l')^c = l''$ using comp-exi2[of $l''$] by auto

from this $l''$-p have $l'$-p:$\forall l \in L^C. l \cdot \sigma = (l')^c$ by auto

have $\forall l \in L. l \cdot \sigma = l'$

proof

fix $l$

assume $l \in L$

then have $l^c \in L^C$ by auto

then have $(l^c) \cdot \sigma = (l')^c$ using $l'$-p by auto

then have $(l \cdot \sigma)^c = (l')^c$ by (cases $l$) auto

then show $l \cdot \sigma = l'$ using cancel-comp by blast

qed

then show $\unifier_{ts} \sigma L$ unfolding $\unifier_{ts}$-def by auto

next

assume $\unifier_{ts} \sigma L$

then obtain $l'$ where $l'$-p: $\forall l \in L. l \cdot \sigma = l'$ unfolding $\unifier_{ts}$-def by auto

have $\forall l \in L^C. l \cdot \sigma = (l')^c$

proof

fix $l$

assume $l \in L^C$

then have $l^c \in L$ using cancel-comp1 by (metis image-iff)

then show $(l \cdot \sigma)^c = (l')^c$ using $l'$-p comp-sub cancel-comp1 by metis

qed

then show $\unifier_{ts} \sigma (L^C)$ unfolding $\unifier_{ts}$-def by auto

qed

**Lemma**

unifier-sub1: $\unifier_{ts} \sigma L \implies L' \subseteq L \implies \unifier_{ts} \sigma L'$

unfolding $\unifier_{ts}$-def by auto

**Lemma**

unifier-sub2:

assumes asm: $\unifier_{ts} \sigma (L_1 \cup L_2)$

shows $\unifier_{ts} \sigma L_1 \land \unifier_{ts} \sigma L_2$

proof

have $L_1 \subseteq (L_1 \cup L_2) \land L_2 \subseteq (L_1 \cup L_2)$ by simp

from this asm show $\？thesis$ using unifier-sub1 by auto

qed

9.1 Most General Unifiers

**Definition**

mguts $::=$ substitution $\Rightarrow$ fterm set $\Rightarrow$ bool

where
\[ \text{mgu}_1 \sigma \; \text{ts} \leftrightarrow \text{unifier}_1 \sigma \; \text{ts} \land (\forall u. \; \text{unifier}_1 u \; \text{ts} \rightarrow (\exists i. \; u = \sigma \cdot i)) \]

**Definition**: \( \text{mgu}_1 \sigma \; L \leftrightarrow \text{unifier}_1 \sigma \; L \land (\forall u. \; \text{unifier}_1 u \; L \rightarrow (\exists i. \; u = \sigma \cdot i)) \)

## 10 Resolution

**Definition**: \( \text{applicable} :: \text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{fterm literal set} \Rightarrow \text{fterm literal set} \Rightarrow \text{substitution} \Rightarrow \text{bool} \) where

\[
\text{applicable} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma \leftrightarrow
\begin{align*}
C_1 & \neq \{\} \land C_2 \neq \{\} \land L_1 \neq \{\} \land L_2 \neq \{} \\
\land \; \text{vars}_{\text{ts}}(C_1 \cap \text{vars}_{\text{ts}}(C_2)) = \{\} \\
\land \; L_1 \subseteq C_1 \land L_2 \subseteq C_2 \\
\land \; \text{mgu}_{\text{ts}}(\sigma \cdot (L_1 \cup L_2))
\end{align*}
\]

**Definition**: \( \text{mresolution} :: \text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{fterm literal set} \Rightarrow \text{fterm literal set} \Rightarrow \text{substitution} \Rightarrow \text{fterm clause} \) where

\[
\text{mresolution} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma = ((C_1 \cdot \text{ts} \cdot \sigma) - (L_1 \cdot \text{ts} \cdot \sigma)) \cup ((C_2 \cdot \text{ts} \cdot \sigma) - (L_2 \cdot \text{ts} \cdot \sigma))
\]

**Definition**: \( \text{resolution} :: \text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{fterm literal set} \Rightarrow \text{fterm literal set} \Rightarrow \text{substitution} \Rightarrow \text{fterm clause} \) where

\[
\text{resolution} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot \text{ts} \cdot \sigma
\]

**Inductive**: \( \text{mresolution-step} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool} \) where

\[
\text{mresolution-rule}: \quad C_1 \in \text{Cs} \quad \Rightarrow \quad C_2 \in \text{Cs} \quad \Rightarrow \quad \text{applicable} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma \quad \Rightarrow \quad \text{mresolution-step} \; \text{Cs} \; (\text{Cs} \cup \{\text{mresolution} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma\})
\]

|standardize-apart:
\[
C \in \text{Cs} \quad \Rightarrow \quad \text{var-renaming-of} \; C \; C' \quad \Rightarrow \quad \text{mresolution-step} \; \text{Cs} \; (\text{Cs} \cup \{C'\})
\]

**Inductive**: \( \text{resolution-step} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool} \) where

\[
\text{resolution-rule}: \quad C_1 \in \text{Cs} \quad \Rightarrow \quad C_2 \in \text{Cs} \quad \Rightarrow \quad \text{applicable} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma \quad \Rightarrow \quad \text{resolution-step} \; \text{Cs} \; (\text{Cs} \cup \{\text{resolution} \; C_1 \; C_2 \; L_1 \; L_2 \; \sigma\})
\]

|standardize-apart:
\[
C \in \text{Cs} \quad \Rightarrow \quad \text{var-renaming-of} \; C \; C' \quad \Rightarrow \quad \text{resolution-step} \; \text{Cs} \; (\text{Cs} \cup \{C'\})
\]

**Definition**: \( \text{mresolution-deriv} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool} \) where

\[
\text{mresolution-deriv} = \text{rtranclp} \; \text{mresolution-step}
\]

**Definition**: \( \text{resolution-deriv} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool} \) where

\[
\text{resolution-deriv} = \text{rtranclp} \; \text{resolution-step}
\]

39
11 Soundness

definition evalsub :: 'a var-denot ⇒ 'a fun-denot ⇒ substitution ⇒ 'a var-denot
where
  evalsub E F σ = eval₁ E F ◦ σ

lemma substitutiont: eval₁ E F (t ·₁ σ) = eval₁ (evalsub E F σ) F t
apply (induction t)
unfolding evalsub-def apply auto
apply (metis (mono-tags, lifting) comp-apply map-cong)
done

lemma substitutiont: eval₁ E F (ts ·₁ ts σ) = eval₁ (evalsub E F σ) F ts
using substitutiont by auto

done

lemma subst-sound:
  assumes asm: evalₐ F G C
  shows evalₐ F G (C ·ₐ σ)
proof −
  have ∀ E. ∃ l ∈ C ·ₐ σ. evalₐ E F G l
  proof
    fix E
    from asm have ∀ E. ∃ l ∈ C. evalₐ E F G l unfolding evalₐ-def by auto
then have ∃ l ∈ C. evalₐ (evalsub E F σ) F G l by auto
then show ∃ l ∈ C ·ₐ σ. evalₐ E F G l using substitution by blast
  qed
then show evalₐ F G (C ·ₐ σ) unfolding evalₐ-def by auto
  qed

lemma simple-resolution-sound:
  assumes C₁sat: evalₐ F G C₁
  assumes C₂sat: evalₐ F G C₂
  assumes l₁inc₁: l₁ ∈ C₁
  assumes l₂inc₂: l₂ ∈ C₂
  assumes comp: l₁ = l₂
  shows evalₐ F G ((C₁ − {l₁}) ∪ (C₂ − {l₂}))
proof −
  have ∀ E. ∃ l ∈ (((C₁ − {l₁}) ∪ (C₂ − {l₂})). evalₐ E F G l
  proof
    fix E
    have evalₐ E F G l₁ ∨ evalₐ E F G l₂ using comp by (cases l₁) auto
then show ∃ l ∈ (((C₁ − {l₁}) ∪ (C₂ − {l₂})). evalₐ E F G l
    proof
 assume eval₁ E F G l₁ 
  then have ¬eval₁ E F G l₂ using comp by (cases l₁) auto 
  then have ∃l₂′ ∈ C₂. l₂′ ≠ l₂ ∧ eval₁ E F G l₂′ using l₂inc₂ C₂sat 
 unfolding evalc-def by auto 
  then show ∃l ∈ (C₁ - {l₁}) ∪ (C₂ - {l₂}). eval₁ E F G l by auto 
 next 
  assume eval₁ E F G l₂ 
  then have ¬eval₁ E F G l₁ using comp by (cases l₁) auto 
  then have ∃l₁′ ∈ C₁. l₁′ ≠ l₁ ∧ eval₁ E F G l₁′ using l₁inc₁ C₁sat 
 unfolding evalc-def by auto 
  then show ∃l ∈ (C₁ - {l₁}) ∪ (C₂ - {l₂}). eval₁ E F G l by auto 
 qed 
 qed 
 then show ?thesis unfolding evalc-def by simp 
 qed 

 lemma mresolution-sound: 
  assumes sat₁: evalc E F G C₁ 
  assumes sat₂: evalc E F G C₂ 
  assumes appl: applicable C₁ C₂ L₁ L₂ σ 
  shows evalc E F G (mresolution C₁ C₂ L₁ L₂ σ) 
 proof – 
  from sat₁ have sat₁σ: evalc E F G (C₁ ∪σ σ) using subst-sound by blast 
  from sat₂ have sat₂σ: evalc E F G (C₂ ∪σ σ) using subst-sound by blast 

 from appl obtain l₁ where l₁-p: l₁ ∈ L₁ unfolding applicable-def by auto 

 from l₁-p appl have l₁ ∈ C₁ unfolding applicable-def by auto 
  then have inc₁σ: l₁ ∪ σ ∈ C₁ ∪σ σ by auto 
 from l₁-p have unified₁: l₁ ∈ (L₁ ∪ (L₂ − L₁)) by auto 
 from l₁-p appl have l₁σisl₁σ: {l₁ ∪ σ} = L₁ ∪σ σ 
    unfolding mgu₁σ-def unifier₁σ-def applicable-def by auto 
 from appl obtain l₂ where l₂-p: l₂ ∈ L₂ unfolding applicable-def by auto 

 from l₂-p appl have l₂ ∈ C₂ unfolding applicable-def by auto 
  then have inc₂σ: l₂ ∪ σ ∈ C₂ ∪σ σ by auto 
 from l₂-p have unified₂: l₂ ∈ (L₁ ∪ (L₂ − L₁)) by auto 
 from unified₁ unified₂ appl have l₁ ∪ σ = (l₂ − C) ∪ σ 
    unfolding mgu₁σ-def unifier₁σ-def applicable-def by auto 
  then have comp: (l₁ ∪ σ)c = l₂ ∪ σ using comp-sub comp-swap by auto 
 from appl have unifier₁σ (L₂ − C) 
    using unifier-sub2 unfolding mgu₁σ-def applicable-def by blast 
  then have unifier₁σ L₂ by auto
from this l₂-p have \( l₂σ \cdot l₂σ = L₂ \cdot l₂σ \) unfolding unifier₁σ-def by auto

from sat₁σ sat₂σ inc₁σ inc₂σ comp have eval₁ F G ((C₁ \cdot g₁ σ) \cup \{(C₂ \cdot g₂ σ) \cup \{(l₁ \cdot g₁ σ) \cup \{(l₂ \cdot g₂ σ)\})) using simple-resolution-sound[of F G C₁ \cdot g₁ σ C₂ \cdot g₂ σ l₁ \cdot g₁ σ l₂ \cdot g₂ σ]
by auto
from this l₁σisl₁σ l₂σisl₂σ show ?thesis unfolding mresolution-def by auto qed

lemma resolution-superset: mresolution C₁ C₂ L₁ L₂ σ ⊆ resolution C₁ C₂ L₁ L₂ σ

unfolding mresolution-def resolution-def by auto

lemma superset-sound:
assumes sup: C ⊆ C'
assumes sat: eval₁ F G C
shows eval₁ F G C'
proof –
  have \( \forall E. \exists l \in C'. \text{ eval₁ E F G l} \)
  proof
    fix E
    from sat have \( \forall E. \exists l \in C. \text{ eval₁ E F G l} \) unfolding eval₁-def by –
    then have \( \exists l \in C'. \text{ eval₁ E F G l} \) by auto
    then show \( \exists l \in C'. \text{ eval₁ E F G l} \) using sup by auto
  qed
  then show eval₁ F G C' unfolding eval₁-def by auto
qed

lemma resolution-sound:
assumes sat₁: eval₁ F G C₁
assumes sat₂: eval₁ F G C₂
assumes appl: applicable C₁ C₂ L₁ L₂ σ
shows eval₁ F G (resolution C₁ C₂ L₁ L₂ σ)
proof –
  from sat₁ sat₂ appl have eval₁ F G (mresolution C₁ C₂ L₁ L₂ σ) using mresolution-sound by blast
  then show ?thesis using superset-sound resolution-superset by metis
qed

lemma sound-step: mresolution-step Cs Cs' \n\Rightarrow\ eval₁cs F G Cs \n\Rightarrow\ eval₁cs F G Cs'

proof (induction rule: mresolution-step.induct)
case (mresolution-rule C₁ Cs C₂ l₁ l₂ σ)
  then have eval₁ F G C₁ \land eval₁ F G C₂ unfolding eval₁cs-def by auto
  then have eval₁ F G (mresolution C₁ C₂ l₁ l₂ σ)
    using mresolution-sound mresolution-rule by auto
  then show ?case using mresolution-rule unfolding eval₁cs-def by auto
next
case (standardize-apart C Cs C')
then have evalc F G C unfolding evalcs-def by auto
then have evalc 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 evalcs-def by auto
qed

lemma lsound-step: resolution-step Cs Cs' =⇒ evalcs F G Cs =⇒ evalcs F G Cs'
proof (induction rule: resolution-step.induct)
case (resolution-rule C1 Cs C2 l1 l2 σ)
then have evalc F G C1 ∧ evalc F G C2 unfolding evalcs-def by auto
then have evalc F G (resolution C1 C2 l1 l2 σ)
using resolution-sound resolution-rule by auto
then show ?case using resolution-rule unfolding evalcs-def by auto
next
case (standardize-apart C Cs C')
then have evalc F G C unfolding evalcs-def by auto
then have evalc 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 evalcs-def by auto
qed

lemma sound-derivation: mresolution-deriv Cs Cs' =⇒ evalcs F G Cs =⇒ evalcs 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 Cs1 Cs2 Cs3) then show ?case using sound-step by auto
qed

lemma lsound-derivation: resolution-deriv Cs Cs' =⇒ evalcs F G Cs =⇒ evalcs 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 Cs1 Cs2 Cs3) 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: ground t \implies (eval t \ HFun t) = hterm-of-fterm t
  by (induction t) auto

lemma eval-ground ts: ground ts \implies (eval ts \ HFun ts) = hterms-of-fterms ts
  unfolding hterms-of-fterms-def using eval-ground, by (induction ts) auto

lemma eval-l-ground ts:
  assumes asm: ground ts
  shows eval l \ E \ HFun G (Pos P ts) \iff G P (hterms-of-fterms ts)
proof
  have eval l \ E \ HFun G (Pos P ts) = G P (eval ts \ 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 \iff
    ground l
    \land (let i = nat-from-fatom (get-atom l) in
      i < length G \land G ! i = (\neg \text{sign l})
    )

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

abbreviation falsifies G C :: partial-pred-denot \Rightarrow fterm clause \Rightarrow bool where
  falsifies G C \equiv (\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'. \text{instance-of ts} C' C \land falsifies 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 l (t \cdot, (sub-of-denot E))
by (induction t) (auto simp add: ground-fterm-of-hterm)

lemma ground-sub-of-denotts: ground\_t\_s (ts \cdot ts sub-of-denot E)
using ground-sub-of-denott by simp

lemma ground-sub-of-denottl: ground\_l (l \cdot l sub-of-denot E)
proof
  have ground\_t\_s (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 = eval\_\_t E HFun t
using sub-of-denot-equivx by (induction t) auto

lemma sub-of-denot-equivts: eval\_\_t E HFun (ts \cdot ts (sub-of-denot E)) = eval\_\_t E HFun ts
using sub-of-denot-equivt by simp

lemma sub-of-denot-equivl: eval\_\_t E HFun G (l \cdot l sub-of-denot E) \iff eval\_\_t E HFun G l
proof (induction l)
  case (Pos p ts)
  have eval\_\_t E HFun G ((Pos p ts) \cdot sub-of-denot E) \iff G p (eval\_\_t E HFun (ts \cdot ts (sub-of-denot E))) by auto
  also have ... \iff G p (eval\_\_t E HFun ts) using sub-of-denot-equivts[of E ts]
  by metis
  also have ... \iff eval\_\_t E HFun G (Pos p ts) by simp
  finally show ?case by blast
next
  case (Neg p ts)
  have eval\_\_t E HFun G ((Neg p ts) \cdot sub-of-denot E) \iff \neg G p (eval\_\_t E HFun (ts \cdot ts (sub-of-denot E))) by auto
  also have ... \iff \neg G p (eval\_\_t E HFun ts) using sub-of-denot-equivts[of E ts]
by metis
also have \( \ldots = \text{eval}_l \ E \ HFun \ G (\text{Neg} \ p \ ts) \) by simp
finally
show \(?\text{case}\) by blast
qed

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

**Lemma sub-of-denot-equiv-ground**:  
\[
eval_l E \ HFun G \ l = \eval_l E \ HFun G \ (l \cdot \eta \ \text{sub-of-denot} \ E) \land \text{ground}_l (l \cdot \eta \ \text{sub-of-denot} \ E)
\]  
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  
shows falsifies\(_c\) G C  
proof –  
from assms obtain C' where C'-p: instance-of\(\_\)s C' \(\cdot\)s \(\tau\) \(\land\) falsifies\(_g\) G C' by auto  
then have instance-of\(\_\)s (C' \(\cdot\)s \(\tau\)) C unfolding instance-of\(\_\)s-def by auto  
then have instance-of\(\_\)s C' C using C'-p instance-of\(\_\)s-trans by auto  
then show \(?\text{thesis}\) 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) \iff (\exists l \in C \cdot l \ \text{sub-of-denot} \ E. \eval_l E \ HFun G \ l))  
\land \text{ground}_l (C \cdot l \ \text{sub-of-denot} \ E)  
using sub-of-denot-equiv-ground' by auto

**Lemma std\(\_\)1-falsifies: falsifies\(_c\) G C\(_1\) \iff falsifies\(_c\) G (std\(_1\) C\(_1\))**  
proof  
assume asm: falsifies\(_c\) G C\(_1\)  
then obtain C\(_g\) where \(\exists\) C\(_g\) C\(_1\) \land falsifies\(_g\) G C\(_g\) by auto  
moreover  
then have \(\exists\) C\(_g\) (std\(_1\) C\(_1\)) using std-apart-instance-of\(\_\)s\(_1\) instance-of\(\_\)s-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: \(\exists\) (std\(_1\) C\(_1\)) unfolding instance-of\(\_\)s-def by auto

46
from asm obtain Cg where instance-of I Cg (std 1 C1) ∧ falsifies G Cg by auto
moreover
then have instance-of I Cg C1 using inst instance-of I trans assms by blast
ultimately
show falsifies C C1 by auto
qed

lemma std2-falsifies: falsifies C C2 ↔ falsifies C (std 2 C2)
proof
assume asm: falsifies C C2
then obtain Cg where instance-of I Cg C2 ∧ falsifies G Cg by auto
moreover
then have instance-of I Cg (std 2 C2) using std-apart-instance-of I 2 instance-of I trans
asm by blast
ultimately
show falsifies C (std 2 C2) by auto
next
assume asm: falsifies C (std 2 C2)
then have inst: instance-of I (std 2 C2) C2 unfolding instance-of I-def by auto

from asm obtain Cg where instance-of I Cg (std 2 C2) ∧ falsifies G Cg by auto
moreover
then have instance-of I Cg C2 using inst instance-of I trans assms by blast
ultimately
show falsifies C C2 by auto
qed

lemma std1-renames: var-renaming-of C1 (std 1 C1)
proof
have instance-of I C1 (std 1 C1) using std-apart-instance-of I 1 assms by auto
moreover have instance-of I (std 1 C1) C1 using assms unfolding instance-of I-def by auto
ultimately show var-renaming-of C1 (std 1 C1) unfolding var-renaming-of-def by auto
qed

lemma std2-renames: var-renaming-of C2 (std 2 C2)
proof
have instance-of I C2 (std 2 C2) using std-apart-instance-of I 2 assms by auto
moreover have instance-of I (std 2 C2) C2 using assms unfolding instance-of I-def by auto
ultimately show var-renaming-of C2 (std 2 C2) unfolding var-renaming-of-def by auto
qed
14 Semantic Trees

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

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

**definition** closed-tree :: tree ⇒ fterm clause set ⇒ bool where
  closed-tree T Cs ←→ anybranch T (λb. closed-branch b T Cs)
  ∧ anyinternal T (λp. ¬falsifies_{Cs} p Cs)

15 Herbrand’s Theorem

**lemma** maximum:
  assumes asm: finite C
  shows ∃n :: nat. ∀l ∈ C. f l ≤ n
  **proof**
  from asm show ∀l∈C. f l ≤ (Max (f C)) by auto
  qed

**lemma** extend-preserves-model:
  assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
  assumes C-ground: ground_{ls} C
  assumes C-sat: ¬falsifies_{g} (f (Suc n)) C
  assumes n-max: ∀l∈C. nat-from-fatom (get-atom l) ≤ n
  shows eval_c HFun (extend f) C
  **proof**
  let ?F = HFun
  let ?G = extend f
  { fix E
    from C-sat have ∀C′. (¬instance-of_{ls} C′ C ∨ ¬falsifies_{g} (f (Suc n)) C′) by auto
    then have ¬falsifies_{g} (f (Suc n)) C using instance-of_{ls}-self by auto
    then obtain l where l-p: l∈C ∧ ¬falsifies_{l} (f (Suc n)) l using C-ground by blast
    let ?i = nat-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 eval_l HFun (extend f) l
    **proof** (cases l)
      case (Pos P ts)
      from Pos l-p C-ground have ts-ground: ground_{ts} ts by auto
have \( \neg \text{falsifies}_1 (f \cdot (\text{Suc } n)) \cdot l \) \text{ using } l-p \text{ by } auto

then have \( f \cdot (\text{Suc } n) ! \cdot ?i = \text{True} \)

using j-n \text{ Pos ts-ground empty-subts[of ts]} \text{ unfolding falsifies}_1 \text{-def by } auto

moreover have \( f \cdot (\text{Suc } ?i) ! ?i = f \cdot (\text{Suc } n) ! ?i \)

using f-infpath i-n j-n infpath-length[of f] \text{ ith-in-extension[of f]} \text{ by simp}

ultimately

have \( f \cdot (\text{Suc } ?i) ! ?i = \text{True} \text{ using } Pos \text{ by } auto \)

then have \(?G \cdot P \cdot (\text{hterms-of-fterms ts})\) \text{ using } Pos \text{ by } (simp add: nat-from-fatom-def)

then show \(?thesis \text{ using } eval_1 \text{-ground}_1 [\text{of ts} - ?G \cdot P] \text{ ts-ground Pos by } auto\)

next

case \((\text{Neg P ts})\)

from \(\text{Neg l-p C-ground}\) have \(\text{ts-ground : ground}_1 \cdot ts \cdot by \ auto\)

have \(\neg \text{falsifies}_1 (f \cdot (\text{Suc } n)) \cdot l \) \text{ using } l-p \text{ by } auto

then have \( f \cdot (\text{Suc } n) ! \cdot ?i = \text{False} \)

using j-n \text{ Neg ts-ground empty-subts[of ts]} \text{ unfolding falsifies}_1 \text{-def by } auto

moreover have \( f \cdot (\text{Suc } ?i) ! ?i = f \cdot (\text{Suc } n) ! ?i \)

using f-infpath i-n j-n infpath-length[of f] \text{ ith-in-extension[of f]} \text{ by simp}

ultimately

have \( f \cdot (\text{Suc } ?i) ! ?i = \text{False} \text{ using } Neg \text{ by } auto \)

then have \( \neg ?G \cdot P \cdot (\text{hterms-of-fterms ts})\) \text{ using } Neg \text{ by } (simp add: nat-from-fatom-def)

then show \(?thesis \text{ using } Neg \text{ eval}_1 \text{-ground}_1 [\text{of ts} - ?G \cdot P] \text{ ts-ground } Pos \text{ by } auto\)

qed
lemma extend-infpath:
  assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
  assumes model-c: ∀ n. ¬falsifies C (f n) C
  assumes fin-c: finite C
  shows eval C HFun (extend f) C
unfolding eval C-def proof
  fix E
  let ?G = extend f
  let ?σ = sub-of-denot E
  from fin-c have fin-σ: finite (C · ls ? σ) using sub-of-denot-equiv-ground by auto
  have groundcσ: ground (C · ls ? σ) using groundcσ using sub-of-denot-equiv-ground by auto
  then have eval C HFun ? G (C · ls ? σ) using groundcσ f-infpath fin-c σ extend-preserves-model2[of f C ? σ] by blast
  then have eval C HFun ? G (C · ls ? σ) using groundcσ f-infpath fin-c σ extend-preserves-model2[of f C ? σ] by blast
  then have eval C HFun ? G (C · ls ? σ) using groundcσ f-infpath fin-c σ extend-preserves-model2[of f C ? σ] by blast
  then show eval C HFun ? G (C · ls ? σ) using groundcσ f-infpath fin-c σ extend-preserves-model2[of f C ? σ] by blast
qed

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

lemma infpath-model:
  assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
  assumes model-cs: ∀ n. ¬falsifies Cs (f n) Cs
  assumes fin-cs: finite Cs
  assumes fin-c: ∀ C ∈ Cs. finite C
  shows eval Cs HFun (extend f) Cs
proof –
  let ?F = HFun
  have ∀ C ∈ Cs. eval C ? F (extend f) C
    proof (rule ballI)
      fix C
      assume asm: C ∈ Cs
      then have ∀ n. ¬falsifies Cs (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-falsifies1:
assumes falsifies1 ds l
shows falsifies1 (ds @ d) l
proof –
  let ?i = nat-from-fatom (get-atom l)
  from assms have i-p: ground l ∧ ?i < length ds ∧ ds ! ?i = (¬sign l) unfolding falsifies1-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 falsifies1-def by simp
qed

lemma longer-falsifies2:
assumes falsifies2 ds C
shows falsifies2 (ds @ d) C
proof –
lemma longer-falsifies:
  assumes falsifies ds Cs
  shows falsifies (ds @ d) Cs
proof –
  from assms obtain C where instance-of₁, C' C ∧ falsifies₉ ds C' by auto
  moreover then have falsifies₉ (ds @ d) C' using longer-falsifies₉ by auto
  ultimately show ?thesis by auto
qed

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

lemma longer-falsifies₂:
  assumes falsifies₂ ds Cs
  shows falsifies₂ (ds @ d) Cs
proof –
  from assms obtain C where C ∈ Cs ∧ falsifies₂ ds C by auto
  moreover then have falsifies₂ (ds @ d) C using longer-falsifies₂[of C ds d] by blast
  ultimately show ?thesis by auto
qed

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

theorem herbrand':
  assumes openb: ∀ T. ∃ G. open-branch G T Cs
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  shows ∃ G. eval Cs HFun G Cs
proof –
  — Show T infinite:
  let ?tree = { G. ¬falsifies₂ G Cs}
  let ?undiag = length
  let ?diag = (λl. SOME b. open-branch b (deeptree l) Cs) :: nat ⇒ partial-pred-denot

  from openb have diag-open: ∀ l. open-branch (?diag l) (deeptree l) Cs
    using someI-ex[of λb. open-branch b (deeptree -) Cs] by auto
  then have ∀ n. ?undiag (?diag n) = n using branch-length by auto
  moreover have ∀ n. (?diag n) ∈ ?tree using diag-open by auto
  ultimately have ¬finite ?tree using infinity[of - λn. SOME b. open-branch b (- n) Cs] by simp
— Get infinite path:

moreover

have \( \forall ds \ d. \neg \text{falsifies}_{cs} (ds @ d) \ Cs \rightarrow \neg \text{falsifies}_{cs} ds Cs \)

using longer-falsifies[of Cs] by blast

then have \( (\forall ds \ d. \ ds @ d \in \ ?\text{tree} \rightarrow ds \in \ ?\text{tree}) \) by auto

ultimately

have \( \exists c. \ \text{wf-infpath} c \land (\forall n. \ c \ n \in \ ?\text{tree}) \) using konig[of ?tree] by blast

then have \( \exists G. \ \text{wf-infpath} G \land (\forall n. \ \neg \text{falsifies}_{cs} (G \ n) \ Cs) \) by auto

— Apply above infpath lemma:

then show \( \exists G. \ \text{eval}_{cs} HFun \ G \ Cs \) using infpath-model finite-cs by auto

qed

lemma shorter-falsifies1:

assumes falsifies1 \((ds@d)\ l\)

assumes nat-from-fatom \((\text{get-atom} \ l)\ < \ \text{length} \ ds\)

shows falsifies1 \ds \ l\)

proof

let \( ?i = \text{nat-from-fatom} (\text{get-atom} \ l)\)

from assms have i-p: \( \text{ground} l \land \ ?i < \text{length} (ds@d) \land (ds@d) \ ! \ ?i = (\neg \text{sign} l)\)

unfolding falsifies1-def by meson

moreover

then have \( ?i < \text{length} \ ds \) using assms by auto

moreover

then have \( ds ! \ ?i = (\neg \text{sign} l) \) using i-p nth-append[of ds d ?i] by auto

ultimately show \( \neg \text{thesis} \) using assms unfolding falsifies1-def by simp

qed

theorem herbrand′-contra:

assumes finite-cs: finite Cs \( \forall C \in Cs. \ \text{finite} \ C\)

assumes unsat: \( \forall G. \ \neg \text{eval}_{cs} HFun G Cs\)

shows \( \exists T. \ \forall G. \ \text{branch} G T \rightarrow \text{closed-branch} G T Cs\)

proof

from finite-cs unsat have \( \forall T. \ \exists G. \ \text{open-branch} G T Cs \implies \exists G. \ \text{eval}_{cs} HFun G Cs \) using herbrand′ by blast

then show \( \neg \text{thesis} \) using unsat by blast

qed

theorem herbrand:

assumes unsat: \( \forall G. \ \neg \text{eval}_{cs} HFun G Cs\)

assumes finite-cs: finite Cs \( \forall C \in Cs. \ \text{finite} \ C\)

shows \( \exists T. \ \text{closed-tree} T Cs\)

proof

from unsat finite-cs obtain T where anybranch T (\lambda b. \text{closed-branch} b T Cs) using herbrand′-contra[of Cs] by blast

then have \( \exists T. \ \text{anybranch} T (\lambda p. \ \neg \text{falsifies}_{cs} p Cs) \land \text{anyinternal} T (\lambda p. \ \neg \text{falsifies}_{cs} p Cs)\)

using cutoff-branch-internal[of T \lambda p. falsifies_{cs} p Cs] by blast

then show \( \neg \text{thesis} \) unfolding closed-tree-def by auto

qed
16 Lifting Lemma

theory Completeness imports Resolution begin

locale unification =
  assumes unification: \( \forall \sigma. \text{finite } L \implies \text{unifier}_{ls} \sigma L \implies \exists \vartheta. \text{mgu}_{ls} \vartheta L \) 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: \text{finite } C \land \text{finite } D
  assumes apart: \text{vars}_{ls} C \cap \text{vars}_{ls} D = \{\}
  assumes inst1: \text{instance-of}_{ls} C' C
  assumes inst2: \text{instance-of}_{ls} D' D
  assumes appl: \text{applicable } C' D' L' M' \sigma
  shows \( \exists L M \tau. \text{applicable } C D L M \tau \land \text{instance-of}_{ls} (\text{resolution } C' D' L' M' \sigma) (\text{resolution } C D L M \tau) \)

proof −
  let \(?C'_1 = C' - L'\)
  let \(?D'_1 = D' - M'\)

  from inst1 obtain \(\text{lbnd where } \text{lbnd-p: } C \cdot_{ls} \text{lbnd} = C'\) unfolding instance-of_{ls}-def

  from inst2 obtain \(\mu where \mu-p: D \cdot_{ls} \mu = D'\) unfolding instance-of_{ls}-def

  from \(\mu-p \text{ lbnd-p apart obtain } \eta where \eta-p: C \cdot_{ls} \eta = C' \land D \cdot_{ls} \eta = D'\) using merge-sub by force

  from \(\eta-p \text{ have } \exists L \subseteq C. L \cdot_{ls} \eta = L' \land (C - L) \cdot_{ls} \eta = C'_1\) using appl project-sub[\eta] C C' L' unfolding applicable-def by auto

  then obtain \(L where L-p: L \subseteq C \land L \cdot_{ls} \eta = L' \land (C - L) \cdot_{ls} \eta = C'_1\) by auto

  let \(?C'_1 = C - L\)

  from \(\eta-p \text{ have } \exists M \subseteq D. M \cdot_{ls} \eta = M' \land (D - M) \cdot_{ls} \eta = D'_1\) using appl project-sub[\eta] D D' M' unfolding applicable-def by auto

  then obtain \(M where M-p: M \subseteq D \land M \cdot_{ls} \eta = M' \land (D - M) \cdot_{ls} \eta = D'_1\) by auto

  let \(?D'_1 = D - M\)

  from appl have \(\text{mgui}_{ls} \sigma (L' \cup M^C)\) unfolding applicable-def by auto

  then have \(\text{mgui}_{ls} \sigma ((L \cdot_{ls} \eta) \cup (M \cdot_{ls} \eta)^C)\) using L-p M-p by auto

  then have \(\text{mgui}_{ls} \sigma ((L \cup M^C) \cdot_{ls} \eta)\) using compls-subs subs-union by auto

  then have \(\text{unifier}_{ls} \sigma ((L \cup M^C) \cdot_{ls} \eta)\) unfolding mgui_{ls}-def by auto

end
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 $\tau$ where $\tau$-p: mgu$_{ls}$ $\tau$ ($L \cup M^C$) using unification fin by (meson L-p M-p finite-UnI finite-imageI rev-finite-subset)

then obtain $\varphi$ where $\varphi$-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 $?E = ((C - L) \cup (D - M)) \cdot_s \tau$

have $?E \cdot_s \varphi = (?C_1 \cup ?D_1 \cdot_s (\tau \cdot \varphi))$ using subls-union composition-conseq2ls by auto

also have ... = (?C$'1$ \cup ?D$'1$ \cdot_s (\eta \cdot \sigma))$ using $\varphi$-p by auto

also have ... = ((?C$1$ \cdot_s \eta) \cup (?D$1$ \cdot_s \eta)) \cdot_s \sigma using subls-union composition-conseq2ls by auto

finally have $?E \cdot_s \varphi = ((C' - L') \cup (D' - M')) \cdot_s \sigma$ by auto

then have inst: instance-of$_{ls}$ (resolution $C' D' L' M' \cdot_s \sigma$) (resolution $C D L M \cdot_s \tau$)

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 $\eta$-p by auto
} moreover {
  have $D' \neq \{}$ using appl unfolding applicable-def by auto
  then have $D \neq \{}$ using $\eta$-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 $C D L M \cdot_s \tau$

using apart $L$-p $M$-p $\tau$-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 $l$

55
assume \( l \in C \)
then have \( \text{falsifies}_l \parallel l \) using \textit{assms} by \textit{auto}
then show \( \text{False} \) unfolding \textit{falsifies}_l-def by \textit{(cases \( l \))} \textit{auto}
\textit{qed}
thен show \( \textit{thesis} \) by \textit{auto}
\textit{qed}

**lemma** \textit{falsifies}_{x, empty}:
assumes \( \text{falsifies}_c \parallel C \)
shows \( C = \{\} \)
\textit{proof} –
from \textit{assms} obtain \( C' \) where \( C' \)-p: instance-of\(_t\) \( C' \) \& \( \text{falsifies}_g \parallel C' \) by \textit{auto}
then have \( C' = \{\} \) using \textit{falsifies}_g-empty by \textit{auto}
then show \( C = \{\} \) using \( C' \)-p unfolding instance-of\(_t\)-def by \textit{auto}
\textit{qed}

**lemma** \textit{complements-do-not-falsify'}:
assumes \( l1C1'; l1 \in C1' \)
assumes \( l2C1'; l2 \in C1' \)
assumes \( \text{comp}: l1 = l2^{c} \)
assumes \( \text{falsif}: \text{falsifies}_g G C1' \)
shows \( \text{False} \)
\textit{proof} \textit{(cases \( l1 \))}
\textit{case} \( (\text{Pos} p ts) \)
let \( ?i1 = \text{nat-from-fatom} (p, ts) \)
from \textit{assms} have \( \text{gr}: \text{ground}_t \ l1 \) unfolding \textit{falsifies}_l-def by \textit{auto}
then have \( \text{Neg}: l2 = \text{Neg} p ts \) using \( \text{comp} \ \text{Pos} \) by \textit{(cases \( l2 \))} \textit{auto}

from \( \textit{falsif} \) have \( \text{falsifies}_l G l1 \) using \( l1C1' \) by \textit{auto}
then have \( G ! ?i1 = \text{False} \) using \( l1C1' \text{ Pos unfolding falsifies}_l-def \) by \textit{(induction Pos p ts)} \textit{auto}
moreover
let \( ?i2 = \text{nat-from-fatom} (\text{get-atom} l2) \)
from \( \textit{falsif} \) have \( \text{falsifies}_l G l2 \) using \( l2C1' \) by \textit{auto}
then have \( G ! ?i2 = (\neg \text{sign} l2) \) unfolding \( \text{falsifies}_l\)-def by \textit{meson}
then have \( G ! ?i1 = (\neg \text{sign} l2) \) using \( \text{Pos} \text{ Neg comp by simp} \)
then have \( G ! ?i1 = \text{True} \) using \( \text{Neg} \) by \textit{auto}
ultimately show \( \textit{thesis} \) by \textit{auto}
\textit{next}
\textit{case} \( (\text{Neg} p ts) \)
let \( ?i1 = \text{nat-from-fatom} (p, ts) \)
from \textit{assms} have \( \text{gr}: \text{ground}_t \ l1 \) unfolding \textit{falsifies}_l-def by \textit{auto}
then have \( \text{Pos}: l2 = \text{Pos} p ts \) using \( \text{comp} \ \text{Neg} \) by \textit{(cases \( l2 \))} \textit{auto}

from \( \textit{falsif} \) have \( \text{falsifies}_l G l1 \) using \( l1C1' \) by \textit{auto}
then have \( G ! ?i1 = \text{True} \) using \( l1C1' \ \text{Neg unfolding falsifies}_l\)-def by \textit{(metis}
get-atom.simps(2) literal.disc(2)

moreover
let ?i2 = nat-from-fatom (get-atom l2)
from falsif have falsifies l2 C1' by auto
then have G ! ?i2 = (~sign l2) unfolding falsifies-def by meson
then have G ! ?i1 = (~sign l2) 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': l1 ∈ C1'
assumes l2C1': l2 ∈ C1'
assumes fals: falsifies g G C1'
shows l1 ≠ l2

proof

let ?i = nat-from-fatom (get-atom lo)

have ground-l2: ground(l l using l-p C1'-p by auto
— They are, of course, also ground:
have ground-lo: ground(lo using C1'-p other by auto
from C1'-p have falsifies (B@[d]) (C1' - {l}) by auto
— And indeed, falsified by B @ [d]:
then have loB2: falsifies (B@[d]) lo using other by auto
then have ?i < length (B @[d]) unfolding falsifies-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 ≠ l' using C1'-p l-p other complements-do-not-falsify[of lo C1' 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-l2 ground1-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\_1 B lo using laB\_2 shorter-falsifies\_1 by blast

qed

\textbf{theorem} completeness‘:

\textbf{shows} closed-tree T Cs \implies \forall C \in Cs. finite C \implies \exists Cs'. resolution-deriv Cs Cs' \\
\textbf{\&} \{\} \in Cs'

\textbf{proof} (induction T arbitrary: Cs rule: measure-induct-rule[of treesize])

\textbf{fix} T::tree

\textbf{fix} Cs :: \textit{fterm clause set}

\textbf{assume} \textit{ih}: (\forall T' Cs. treesize T' < treesize T \implies closed-tree T' Cs \implies \forall C \in Cs. finite C \implies \exists Cs'. resolution-deriv Cs Cs' \& \{\} \in Cs')

\textbf{assume} clo: closed-tree T Cs

\textbf{assume} finite-Cs: \forall C \in Cs. finite C

\{ — Base case:

\textbf{assume} treesize T = 0

\textbf{then have} T=Leaf using treesize-Leaf by auto

\textbf{then have} closed-branch \[ Leaf Cs \textbf{using} branch-inv-Leaf clo \textbf{unfolding} closed-tree-def \textbf{by} auto

\textbf{then have} falsifies\_cs \[ Cs \textbf{by} auto

\textbf{then have} \{\} \in Cs using falsifies\_cs\text{-empty \textbf{by} auto

\textbf{then have} \exists Cs'. resolution-deriv Cs Cs' \& \{\} \in Cs' \textbf{unfolding} resolution-deriv-def \textbf{by} auto

\}

moreover

\{ — Induction case:

\textbf{assume} treesize T > 0

\textbf{then have} \exists l r. T=Branching l r by (cases T) auto

— Finding sibling branches and their corresponding clauses:

\textbf{then obtain} B where b-p: internal B T \& branch (B@[True]) T \& branch (B@[False]) T

\textbf{using} internal-branch[of - \[ - T] Branching-Leaf-Leaf-Tree \textbf{by fastforce

\textbf{let} ?B\_1 = B@[True]

\textbf{let} ?B\_2 = B@[False]

\textbf{obtain} C\_1\_o where C\_1\_o-p: C\_1\_o \in Cs \& falsifies\_e \ ?B\_1 C\_1\_o using b-p clo

\textbf{unfolding} closed-tree-def \textbf{by} metis

\textbf{obtain} C\_2\_o where C\_2\_o-p: C\_2\_o \in Cs \& falsifies\_e \ ?B\_2 C\_2\_o using b-p clo

\textbf{unfolding} closed-tree-def \textbf{by} metis

— Standardizing the clauses apart:

\textbf{let} ?C\_1 = std\_1 C\_1\_o

\textbf{let} ?C\_2 = std\_2 C\_2\_o

\textbf{have} C\_1\_p: falsifies\_e \ ?B\_1 ?C\_1 using std\_1-falsifies C\_1\_o-p \textbf{by auto

\textbf{have} C\_2\_p: falsifies\_e \ ?B\_2 ?C\_2 using std\_2-falsifies C\_2\_o-p \textbf{by auto

58
have fin: finite ?C₁ ∧ finite ?C₂ using C₁-o-p C₂-o-p finite-Cs by auto

— We go down to the ground world.
— Finding the falsifying ground instance C₁' of C₁-o-₁s (λx. ε ("1" @ x)), and proving properties about it:

— C₁' is falsified by B @ [True]:
  from C₁-p obtain C₁' where C₁'-p: ground₁s C₁' ∧ instance-of₁s C₁' ?C₁ ∧ falsifiesg B C₁' by metis

have ¬falsifes₁ B C₁-o using C₁-o-p b-p clo unfolding closed-tree-def by metis
then have ¬falsifes₁ B ?C₁ using std₁-falsifies using prod.exhaust-set by blast
— C₁' is not falsified by B:
then have l₁-B: ¬¬falsifes₁ B C₁'-p by auto

— C₁' contains a literal l₁ that is falsified by B @ [True], but not B:
from C₁'-p l₁-B obtain l₁ where l₁-p: l₁ ∈ C₁' ∧ falsifies₁ (B @ [True]) l₁ ∧ ¬falsifies₁ B l₁ by auto
  let ?i = nat-from-fatom (get-atom l₁)

— l₁ is of course ground:
have ground-l₁: ground₁ l₁ using C₁'-p l₁-p by auto

from l₁-p have ¬(∃i < length B ∧ B ! ?i = (¬sign l₁)) using ground-l₁ unfolding falsifies₁-def by meson
then have ¬(∃i < length B ∧ (B @ [True]) ! ?i = (¬sign l₁)) by (metis nth-append) — Not falsified by B.
moreover
from l₁-p have ?i < length (B @ [True]) ∧ (B @ [True]) ! ?i = (¬sign l₁)
unfolding falsifies₁-def by meson
ultimately
have l₁-sign-no: ?i = length B ∧ (B @ [True]) ! ?i = (¬sign l₁) by auto

— l₁ is negative:
from l₁-sign-no have l₁-sign: sign l₁ = False by auto
from l₁-sign-no have l₁-no: nat-from-fatom (get-atom l₁) = length B by auto

— All the other literals in C₁' must be falsified by B, since they are falsified by B @ [True], but not l₁.
from C₁'-p l₁-no l₁-p have B-C₁' l₁: falsifiesg B (C₁' − {l₁}) using other-falsified by blast

— We do the same exercise for C₂-o-₁s (λx. ε ("2" @ x)), C₂', B @ [False], l₂:
  from C₂-p obtain C₂' where C₂'-p: ground₁s C₂' ∧ instance-of₁s C₂' ?C₂ ∧ falsifiesg ?B₂ C₂' by metis

have ¬falsifes₂ B C₂-o using C₂-o-p b-p clo unfolding closed-tree-def by metis
then have ¬falsifies₂ B ?C₂ using std₂-falsifies using prod.exhaust-set by blast
then have l-B: ¬falsifies₂ B C₂' using C₂'-p by auto

— C₂' contains a literal l₂ that is falsified by B @ [False], but not B:
from C₂'-p l-B obtain l₂ where l₂-p: l₂ ∈ C₂' ∧ falsifies₂ (B@[False]) l₂ ∧ ¬falsifies₂ B l₂ by auto
let ?i = nat-from-fatom (get-atom l₂)

have ground-l₂: ground₁ l₂ using C₂'-p l₂-p by auto

from l₂-p have ¬(?i < length B ∧ B ! ?i = (¬sign l₂₂)) using ground-l₂
unfolding falsifies₁-def by meson
then have ¬(?i < length B ∧ (B@[False]) ! ?i = (¬sign l₂)) by (metis nth-append) — Not falsified by B.
moreover from l₂-p have ?i < length (B@[False]) ∧ (B@[False]) ! ?i = (¬sign l₂) unfolding falsifies₁-def by meson
ultimately
have l₂-sign-no: ?i = length B ∧ (B@[False]) ! ?i = (¬sign l₂) by auto
— l₂ is negative:
from l₂-sign-no have l₂-sign: sign l₂ = True by auto
from l₂-sign-no have l₂-no: nat-from-fatom (get-atom l₂) = length B by auto
— All the other literals in C₂' must be falsified by B, since they are falsified by B @ [False], but not l₂.
from C₂'-p l₂-no l₂-p have B-C₂' l₂₂: falsifies₂ B (C₂' \ {l₂})
using other-falsified by meson
— Proving some properties about C₁' and C₂', l₁ and l₂, as well as the resolvent of C₁' and C₂':
have l₂ CIS₁₁: l₂₀ = l₁
proof —
from l₁-no l₂-no ground-l₂ ground-l₂ have get-atom l₁ = get-atom l₂
using nat-from-fatom-bij ground₁-ground-fatom
unfolding bij-btw-def inj-on-def by mesis
then show l₂₀ = l₁ using l₁-sign l₂-sign using sign-comp-atom by mesis
qed

have applicable C₁' C₂' {?l₁} {?l₂} Resolution ε unfolding applicable-def
using l₁-p l₂-p C₁'-p ground₁-vars₁ l₂ CIS₁₁ empty-comp2 unfolding mgu₁₂-def unifier₁₂-def by auto
— Lifting to get a resolvent of C₁ ø ρ₁₁ (λx. ε ("1" @ x)) and C₂ ø ρ₁₂ (λx. ε ("2" @ x)):
then obtain L₁ L₂ τ where L₁ L₂τ-p: applicable ?C₁ ?C₂ L₁ L₂ τ ∧ instance-of₁₁ (resolution C₁' C₂' {?l₁} {?l₂} Resolution ε) (resolution ?C₁ ?C₂ L₁ L₂ τ)
using std-apart-apart C₁'-p C₂'-p lifting[of ?C₁ ?C₂ C₁' C₂' {?l₁} {?l₂}]
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 ?C1 ?C2 L1 L2 \tau by auto
obtain CsNext where CsNext-p: CsNext = Cs \cup \{ ?C1, ?C2, 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_B B ((C1' - \{ l1 \}) \cup (C2' - \{ l2 \})) using B-C1'l1 B-C2'l2 by cases auto
then have falsifies_B B (resolution C1' C2' \{ l1 \} \{ l2 \}) Resolution.ε unfolding resolution-def empty-subs by auto
then have falsifies-C: falsifies_B C using C-p L1L2τ-p by auto

have T''-smaller: treesize T'' < treesize T using treezise-delete T''-p b-p by auto
have T''-bran: anysize T'' (\lambda b. closed-branch b T'' CsNext)
proof (rule allI; rule impI)
fix b
assume br: branch b T''
from br have b = B \lor 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'' (\lambda b. falsifies_cs b CsNext) by auto

— We cut the tree even smaller to ensure only the branches are falsified, i.e. it is a closed tree:
obtain T' where T'-p: T' = cutoff (\lambda G. falsifies_cs G CsNext) [] T'' by auto
have T'-smaller: treesize T' < treesize T using treezise-cutoff[of \lambda G. falsifies_cs G CsNext [] T''] T''-smaller unfolding T''-p by auto

from T''-bran2 have anybranch T' (\lambda b. falsifies_cs b CsNext) using cutoff-branch[of T'' \lambda b. falsifies_cs b CsNext] T'-p by auto
then have T'-bran: anybranch T' (\lambda b. closed-branch b T' CsNext) by auto
have T'-intr: anyinternal T' (\lambda p. \neg falsifies_cs p CsNext) using T'-p cutoff-internal[of T'' \lambda b. falsifies_cs b CsNext] T''-bran2 by blast
have T'-closed: closed-tree T' CsNext using T'-bran T'-intr unfolding

61
closed-tree-def by auto

have finite-CsNext: \( \forall C \in \text{CsNext}. \) 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'' \( \land \{\} \in Cs'' \) using \( \text{ih}[\text{of } T' \text{ CsNext}] \) finite-CsNext by blast
then obtain \( Cs'' \) where Cs''-p: resolution-deriv CsNext Cs'' \( \land \{\} \in Cs'' \) by auto

moreover
\{ — Proving that we can actually derive the new clausal form:
  have resolution-step Cs \((Cs \cup \{?C_1\})\) using \( \text{std}_1\)-renames standardize-apart C1_o-p by (metis Un-insert-right)
moreover
  have resolution-step \((Cs \cup \{?C_1\})\) \((Cs \cup \{?C_1\} \cup \{?C_2\})\) using \( \text{std}_2\)-renames[of C2o] standardize-apart[of C2o - ?C2] C2o-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 \( L_1L_2\tau\)-p resolution-rule[of \(?C_1\) Cs \( \cup \{?C_1,?C_2\}\) \(?C_2\) \(?L_1\) \(?L_2\) \(?\tau\)] 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' \( \land \{\} \in Cs' \) using Cs''-p by auto
ultimately show \( \exists Cs'. \) resolution-deriv Cs Cs' \( \land \{\} \in Cs' \) by auto
qed

theorem completeness:
assumes finite-cs: finite Cs \( \forall C \in Cs. \) finite C
assumes unsat: \( \forall (F::hterm \text{ fun-denot}) \) \((G::hterm \text{ pred-denot})\) . \( -\text{eval}_cs \) \( F \ G \ Cs \)
shows \( \exists Cs'. \) resolution-deriv Cs Cs' \( \land \{\} \in Cs' \)
proof —
  from unsat have \( \forall (G::hterm \text{ pred-denot}) . -\text{eval}_cs \) \( HFun \ G \ Cs \) by auto
  then obtain \( T \) where closed-tree T Cs using herbrand assms by blast
  then show \( \exists Cs'. \) resolution-deriv Cs Cs' \( \land \{\} \in Cs' \) using completeness' assms by auto
qed

end — unification locale
18 Examples

theory Examples imports Resolution begin

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

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

fun G_nat :: nat pred-denot where
  G_nat p [x,y] =
  (if p = "less" ∧ x < y then True else
    if p = "greater" ∧ x > y then True else
    if p = "equals" ∧ x = y then True else False)
| G_nat p us = False

fun E_nat :: nat var-denot where
  E_nat x =
  (if x = "x" then 26 else
    if x = "y" then 5 else 0)

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

end
lemma \eval \E \nat \F \nat \G \nat (\Pos "\greater" [\Var "x", \Var "y'\]) = \text{True} 
by auto

lemma \eval \E \nat \F \nat \G \nat (\Neg "\less" [\Var "x", \Var "y'\]) = \text{True} 
by auto

lemma \eval \E \nat \F \nat \G \nat (\Pos "\less" [\Var "x", \Var "y'\]) = \text{False} 
by auto

lemma \eval \E \nat \F \nat \G \nat (\Pos "\equals" [\Fun "\add" [\Fun "\mult" [\Var "y", \Var "y'\]], \Fun "\one" []] 
, \Var "x'\]) = \text{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_{\text{ls}} \in L \implies \mgu_{\text{ls}} \in L 
unfolding \unifier_{\text{ls}}-def \mgu_{\text{ls}}-def apply auto
apply (rule \text{tac} x=u in exI)
using \empty-comp1 \empty-comp2 apply auto
done

theorem \unifier-single: \unifier_{\text{ls}} \sigma \{ I \}
unfolding \unifier_{\text{ls}}-def by auto

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

lemma \resolution-example1: 
\resolution-deriv \{ \{ \NP, \PQ \}, \{ \NQ \}, \{ \PP, \PQ \} \} 
\{ \{ \NP, \PQ \}, \{ \NQ \}, \{ \PP, \PQ \}, \{ \NP \}, \{ \PP \} \} 

proof – 
have \resolution-step 
\{ \{ \NP, \PQ \}, \{ \NQ \}, \{ \PP, \PQ \} \} 
(\{ \{ \NP, \PQ \}, \{ \NQ \}, \{ \PP, \PQ \} \} \cup \{ \NP \}) 
apply (rule \resolution-rule of \{ \NP, \PQ \} - \{ \NQ \} \{ \PQ \} \{ \NQ \} e) 
unfolding \applicable-def vars_{\text{ls}}-def \ vars_{\text{i}}-def

64
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')
  \{\{NP\}\} - \{\{PP\}\} \{\{NP\}\} ∪ \{PP\}
unfolding applicable-def vars1-def vars1-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\}\}
  \{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}\} ∪ \{\{}\}
apply (rule resolution-rule')
  \{\{NP\}\} - \{\{PP\}\} \{\{NP\}\} ∪ \{PP\}
unfolding applicable-def vars1-def vars1-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\}\}
  \{\{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 \( \text{Nb} :: \text{fterm literal} \) where
\[ \text{Nb} = \text{Neg} "b" [] \]

definition \( \text{Paa} :: \text{fterm literal} \) where
\[ \text{Paa} = \text{Pos} "a" [\text{Fun} "a" []] \]

definition \( \text{Naa} :: \text{fterm literal} \) where
\[ \text{Naa} = \text{Neg} "a" [\text{Fun} "a" []] \]

definition \( \text{Pax} :: \text{fterm literal} \) where
\[ \text{Pax} = \text{Pos} "a" [\text{Var} "x"] \]

definition \( \text{Nax} :: \text{fterm literal} \) where
\[ \text{Nax} = \text{Neg} "a" [\text{Var} "x"] \]

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

lemma \( \text{mguPaaPax-mgu} : \text{mgu} \models \text{mguPaaPax} \) \{\( \text{Paa}, \text{Pax} \}\}

proof
let \( \sigma = \lambda x. \text{if } x = "x" \text{ then Fun } "a" [] \text{ else Var } x \)

have \( a : \text{unifier}_{1s} (\lambda x. \text{if } x = "x" \text{ then Fun } "a" [] \text{ else Var } x) \) \{\( \text{Paa}, \text{Pax} \)\} unfolding \( \text{Paa-def Paa-def unifier}_{1s-def} \) by auto

have \( b : \forall u. \text{unifier}_{1s} u \) \{\( \text{Paa}, \text{Pax} \)\} \( \rightarrow (\exists i. u = ?\sigma \cdot i) \)

proof (rule:rule)
fix \( u \)
assume \( \text{unifier}_{1s} u \) \{\( \text{Paa}, \text{Pax} \)\}
then have \( \text{ww}: u "x" = \text{Fun } "a" [] \) unfolding \( \text{unifier}_{1s-def Paa-def Pax-def} \) by auto

have \( ?\sigma \cdot u = u \)

proof
fix \( x \)
\{ assume \( x="x" \)
moreover
have \( (?\sigma \cdot u) "x" = \text{Fun } "a" [] \) unfolding \( \text{composition-def} \) by auto
ultimately have \( (?\sigma \cdot u) x = u x \) using \( \text{ww} \) by auto
\}
moreover
\{
assume \( x\neq "x" \)
then have \( (?\sigma \cdot u) x = (\varepsilon x) \cdot u \) unfolding \( \text{composition-def} \) 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

66
qed

from a b show ?thesis unfolding mguPaaPax-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\}\}}\}

proof –

have resolution-step
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}\}
apply (rule resolution-rule[of \{Pax\} - \{Na,Pb,Naa\} \{Pax\} \{Naa\} mguPaaPax])

using mguPaaPax-mgu unfolding applicable-def varsP-def varsP-definition

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\}\}}
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}\}
apply (rule resolution-rule[of \{Nb,Na\} - \{Na,Pb\} \{Nb\} \{Pb\} ε])

unfolding applicable-def varsP-definition

Ps-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\}\}}
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}\}

by ( simp add: insert-commute)

moreover

have resolution-step
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}
\{
\{{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}}\}
apply (rule resolution-rule[of \{Na\} - \{Pa\} \{Na\} \{Pa\} ε])

unfolding applicable-def varsP-definition

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 \$\text{thesis}\$ by auto
qed

lemma ref-sound:
  assumes deriv: resolution-deriv Cs Cs' \& \{\} \in Cs'
  shows \$\neg\text{eval}_{cs} F G Cs\$ 
proof --
  from deriv have \$\text{eval}_{cs} F G Cs \implies\text{eval}_{cs} F G Cs'\$ using lsound-derivation by auto
  moreover
  from deriv have \$\text{eval}_{cs} F G Cs' \implies\text{eval}_{c} F G \{\}$ unfolding eval-c-def by auto
  moreover
  then have \$\text{eval}_{c} F G \{\} \implies False\$ unfolding eval-c-def by auto
ultimately show \$\text{thesis}\$ by auto
qed

lemma resolution-example1-sem: \$\neg\text{eval}_{cs} F G \{\{NP, PQ\}, \{N\Q\}, \{PP, PQ\}\}\$
  using resolution-example1 ref-sound by auto

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

end

References


