The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

August 16, 2018

Abstract

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

Contents

1 Terms and Literals 3
   1.1 Ground ................................................. 3
   1.2 Auxiliary ............................................. 4
   1.3 Conversions .......................................... 4
       1.3.1 Conversions - Terms and Herbrand Terms ........ 4
       1.3.2 Conversions - Literals and Herbrand Literals ..... 5
       1.3.3 Conversions - Atoms and Herbrand Atoms ....... 6
   1.4 Enumerations ......................................... 7
       1.4.1 Enumerating Strings .............................. 7
       1.4.2 Enumerating Herbrand Atoms .................... 8
       1.4.3 Enumerating Ground Atoms ..................... 9

2 Trees ...................................................... 9
   2.1 Sizes .................................................. 9
   2.2 Paths ............................................... 10
   2.3 Branches ............................................. 11
1 Terms and Literals

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

type-synonym var-sym = string
type-synonym fun-sym = string
type-synonym pred-sym = string

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

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

type-synonym ′t atom = pred-sym * ′t list

datatype ′t literal =
  sign: Pos (get-pred: pred-sym) (get-terms: ′t list)
| Neg (get-pred: pred-sym) (get-terms: ′t list)

fun get-atom :: ′t literal ⇒ ′t atom where
  get-atom (Pos p ts) = (p, ts)
| get-atom (Neg p ts) = (p, ts)

1.1 Ground

fun ground :: fterm ⇒ bool where
  ground (Var x) ←→ False
| ground (Fun f ts) ←→ (∀ t ∈ set ts. ground t)

abbreviation groundts :: fterm list ⇒ bool where
  groundts ts ≡ (∀ t ∈ set ts. ground t)

abbreviation groundl :: fterm literal ⇒ bool where
  groundl l ≡ groundts (get-terms l)

abbreviation groundls :: fterm literal set ⇒ bool where
  groundls C ≡ (∀ l ∈ C. groundl l)

definition ground-fatoms :: fterm atom set where
  ground-fatoms ≡ {a. groundts (snd a)}

lemma groundl-ground-fatom:
  assumes ground l
  shows get-atom l ∈ ground-fatoms
  using assms unfolding ground-fatoms-def by (induction l) auto
1.2 Auxiliary

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

lemma inv-into-f-f:
  assumes bij-betw f A B
  assumes a ∈ A
  shows (inv-into A f) (f a) = a
using assms bij-betw-inv-into-left by metis

lemma f-inv-into-f:
  assumes bij-betw f A B
  assumes b ∈ B
  shows f ((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 ⇒ fterm where
  fterm-of-hterm (HFun p ts) = Fun p (map fterm-of-hterm ts)

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

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

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

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

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

lemma fterm-of-hterm-hterm-of-fterm[simp]:
assumes ground, t
shows fterm-of-hterm (hterm-of-fterm t) = t
using assms by (induction t) (auto simp add: map-idI)

**Lemma** fterms-of-hterms-hterms-of-fterms[simp]:
assumes ground, ts
shows fterms-of-hterms (hterms-of-fterms ts) = ts
using assms unfolding fterms-of-hterms-def hterms-of-fterms-def by (simp add: map-idI)

**Lemma** ground-fterm-of-hterm: ground, fterm-of-hterm t
by (induction t) (auto simp add: map-idI)

**Lemma** unfolding ground-fterms-of-hterms: ground, (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 (hterms-of-fterms ts)
| flit-of-hlit (Neg p ts) = Neg p (hterms-of-fterms 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, (flit-of-hlit l)
by (induction l) (simp add: ground-fterms-of-hterms)+

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

**Theorem** flit-of-hlit-hlit-of-flit [simp]:
assumes ground, l
shows flit-of-hlit (hlit-of-flit l) = l
using assms by (cases l) auto

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

**Lemma** hlit-of-flit-bij: bij-betw hlit-of-flit {l. ground, l} UNIV
unfolding bij-betw-def
proof
show inj-on hlit-of-flit {l. ground, l} using inj-on-inversel flit-of-hlit-hlit-of-flit
  by (metis (mono-ta, lifting) mem-Collect-eq)
next
have ∀l. ∃l'. ground, 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} = UNIV by auto
qed
lemma flit-of-hlit-bij: bij-betw flit-of-hlit UNIV {\lf. ground \lf}\n
unfolding bij-betw-def inj-on-def

proof
  show \(\forall x \in \text{UNIV}. \forall y \in \text{UNIV}. \text{flit-of-hlit } x = \text{flit-of-hlit } y \rightarrow x = y\)
    using ground-flit-of-hlit hlit-of-flit flit-of-hlit by metis

next
  have \(\forall l. \text{ground } l \rightarrow (l = \text{flit-of-hlit } (\text{hlit-of-flit } l))\)
    using hlit-of-flit flit-of-hlit by auto

moreover
  have \(\forall l. \text{ground } l \subseteq \text{flit-of-hlit } \text{hlit-of-flit } \text{flit-of-hlit}\)
    by blast

ultimately show \(\text{flit-of-hlit } \text{hlit-of-flit } \text{flit-of-hlit} = \{l \in \text{UNIV} | l = \text{flit-of-hlit } \text{hlit-of-flit } \text{flit-of-hlit}\}\)
  using hlit-of-flit flit-of-hlit by auto

qed

1.3.3 Conversions - Atoms and Herbrand Atoms

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

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

lemma ground-fatom-of-hatom: \(\text{ground}_{\text{fatoms }} (\text{snd } (\text{fatom-of-hatom } a))\)
  by (induction a) (simp add: ground-fterms-of-hterms)+

theorem hatom-of-fatom-fatom-of-hatom [simp];
  hatom-of-fatom \((\text{fatom-of-hatom } l) = l\)
  by (cases l) auto

theorem fatom-of-hatom-hatom-of-fatom [simp];
  assumes \(\text{ground}_{\text{fatoms }} (\text{snd } l)\)
  shows \(\text{fatom-of-hatom } (\text{hatom-of-fatom } l) = l\)
  using assms by (cases l) auto

lemma hatom-of-fatoms-bij: bij-betw hatom-of-fatoms UNIV
unfolding bij-betw-def

proof
  show inj-on hatom-of-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 \(\forall a. \exists a'. \text{ground}_{\text{fatoms }} (\text{snd } a') \land a = \text{hatom-of-fatom } a'\)
    using ground-fatom-of-hatom hatom-of-fatoms-fatom-of-hatom by metis
  then show \(\text{hatom-of-fatom } \text{fatoms } = \text{UNIV}\) unfolding ground-fatoms-def
    by blast

qed
lemma fatom-of-hatom-bij: bij_betw fatom-of-hatom UNIV ground-fatoms
unfolding bij_betw_def inj_on_def
proof
  show ∀ x ∈ UNIV. ∀ y ∈ UNIV. fatom-of-hatom x = fatom-of-hatom y ⟷ x = y
    using ground-fatom-of-hatom hatom-of-fatom-fatom-of-hatom by metis
next
  have ∀ a. ground_isa (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_isa (snd (fatom-of-hatom l))
      using ground-fatom-of-hatom by auto
    ultimately show fatom-of-hatom ' UNIV = ground-fatoms
      using hatom-of-fatom-fatom-of-hatom ground-fatom-of-hatom unfolding
      ground-fatoms_def by auto
qed

1.4 Enumerations
1.4.1 Enumerating Strings

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

lemma nat-of-string-bij: bij nat-of-string
proof
  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-of-string_def using someI by metis
qed

lemma string-of-nat-bij: bij string-of-nat unfolding string-of-nat_def using
  nat-of-string-bij bij_betw_inv_into by auto

lemma nat-of-string_string-of-nat[simp]: nat-of-string (string-of-nat n) = n
  unfolding string_of_nat_def
  using nat-of-string_bij f_inv_into_f[of nat-of-string] by simp

lemma string_of_nat_string_of_nat[simp]: string-of-nat (nat-of-string n) = n
  unfolding string_of_nat_def
  using nat_of_string_bij inv_into_f_f[of nat-of-string] by simp
1.4.2 Enumerating Herbrand Atoms

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

\[
\text{nat-of-hatom} \equiv (\text{SOME } f. \text{bij } f)
\]

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

\[
\text{hatom-of-nat} \equiv \text{inv nat-of-hatom}
\]

**instantiation** hterm :: countable begin

instance by countable-datatype

end

**lemma** infinite-hatoms: infinite \((\text{UNIV} :: (\forall t. \text{atom})) \text{ set}\)

proof —

let \(?\text{diago} = \lambda n. (\text{string-of-nat } n,[]))

let \(?\text{undiago} = \lambda a. \text{nat-of-string} (\text{fst } a)

have \(\forall n. \text{?undiago (?\text{diago } n)} = n\) using \(\text{nat-of-string-string-of-nat by auto}

moreover

have \(\forall n. \text{?diago } n \in \text{UNIV}\) by auto

ultimately show infinite \((\text{UNIV} :: (\forall t. \text{atom})) \text{ set}\) using \(\text{infinity[of ?\text{undiago \text{?diago \text{UNIV}}}]}\) by simp

qed

**lemma** nat-of-hatom-bij: bij nat-of-hatom

proof —

let \(?S = \text{UNIV :: (}\forall t. \text{countable} \text{ atom}) \text{ set}\)

have countable \(?S\) by auto

moreover

have infinite \(?S\) using \(\text{infinite-hatoms}\) by auto

ultimately

obtain \(x\) where bij \((x :: \text{hterm atom} ⇒ nat)\) using \(\text{countableE-infinite[of } ?S]\)

by blast

then have bij nat-of-hatom unfolding nat-of-hatom-def using someI by metis

then show ?thesis unfolding bij-betw-def inj-on-def unfolding nat-of-hatom-def

by simp

qed


bij-betw-inv-into by auto

**lemma** nat-of-hatom-hatom-of-nat[simp]: nat-of-hatom (hatom-of-nat \(n\)) = \(n\)

unfolding hatom-of-nat-def

using nat-of-hatom-bij f-inv-into-f[of nat-of-hatom] by simp

**lemma** hatom-of-nat-nat-of-hatom[simp]: hatom-of-nat (nat-of-hatom \(l\)) = \(l\)

unfolding hatom-of-nat-def

using nat-of-hatom-bij inv-into-f[f[of nat-of-hatom - UNIV] by simp
1.4.3 Enumerating Ground Atoms

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

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

**theorem** diag-undiag-fatom[simp]:
assumes ground, ts
shows fatom-of-nat (nat-of-fatom (p, ts)) = (p, ts)
using assms unfolding fatom-of-nat-def nat-of-fatom-def by auto

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

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

**lemma** ground-fatom-of-nat: ground, ts (snd (fatom-of-nat x)) unfolding fatom-of-nat-def
using ground-fatom-of-hatom by auto

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

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
\[ \text{treesize} (\text{Branching } l \ r) = 1 + \text{treesize } l + \text{treesize } r \]

**lemma** treesize-Leaf:
- **assumes** treesize \( T = 0 \)
- **shows** \( T = \text{Leaf} \)
- **using** assms by (cases \( T \)) auto

**lemma** treesize-Branching:
- **assumes** treesize \( T = \text{Suc } n \)
- **shows** \( \exists l. r. \ T = \text{Branching } l \ r \)
- **using** assms by (cases \( T \)) auto

### 2.2 Paths

**fun** path :: dir list ⇒ tree ⇒ bool **where**
- \( \text{path } [] \ T \leftarrow \text{True} \)
- \( \text{path } (d\#ds) \ (\text{Branching } T1 \ T2) \leftarrow (\text{if } d \text{ then path } ds \ T1 \text{ else path } ds \ T2) \)
- \( \text{path } - - \leftarrow \text{False} \)

**lemma** path-inv-Leaf: \( \text{path } p \ \text{Leaf} \leftarrow p = [] \)
- **by** (induction \( p \)) auto

**lemma** path-inv-Cons: \( \text{path } (a\#ds) \ T \rightarrow (\exists l. r. \ T = \text{Branching } l \ r) \)
- **by** (cases \( T \)) (auto simp add: path-inv-Leaf)

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

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

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


qed

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

2.3 Branches

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

lemma has-branch: ∃ b. branch b T
proof (induction T)
case (Leaf)
  have branch [] Leaf by auto
  then show ?case by blast
next
case (Branching 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) \iff branch b l
by auto

lemma branch-inv-Branching-Right:
branch (Right#b) (Branching l r) \iff branch b r
by auto

lemma branch-inv-Branching:
branch b (Branching l r) \iff
(\exists a b'. b = a # b' \land (a \rightarrow branch b' l) \land (\neg a \rightarrow branch b' r))
by (induction b) auto

lemma branch-inv-Leaf2:
T = Leaf \iff (\forall b. branch b T \rightarrow b = [])
proof -
{ assume T=Leaf
then have \forall b. branch b T \rightarrow b = [] using branch-inv-Leaf by auto
}
moreover
{ assume \forall b. branch b T \rightarrow b = []
then have \forall b. branch b T \rightarrow (\exists a b'. b = a # b') by auto
then have \forall b. branch b T \rightarrow (\exists l r. branch b (Branching l r))
using branch-inv-Branching by auto
then have T=Leaf using has-branch[of T] by (metis branch.elims(2))
}
ultimately show T = Leaf \iff (\forall b. branch b T \rightarrow b = []) by auto qed

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

lemma Branching-Leaf-Leaf-Tree:
assumes T = Branching T_1 T_2
shows (\exists B. branch (B@[True]) T \land branch (B@[False]) T)

12
using assms proof (induction T arbitrary: T1 T2)
case Leaf then show ?case by auto
next
case (Branching T1' T2')
{
  assume T1'=Leaf ∧ T2'=Leaf
  then have branch ([[] @ [True]]) (Branching T1' T2') ∧ branch ([[] @ [False]])
    (Branching T1' T2') by auto
    then have ?case by auto
}
moreover
{
  fix T11 T12
  assume T1' = Branching T11 T12
  then obtain B where branch (B @ [True]) T1'
    ∧ branch (B @ [False]) T1' using Branching by blast
  then have branch (([[True] @ B] @ [True]) (Branching T1' T2')
    ∧ branch (([[False] @ B] @ [False]) (Branching T1' T2') by auto
    then have ?case by blast
}
moreover
{
  fix T11 T12
  assume T2' = Branching T11 T12
  then obtain B where branch (B @ [True]) T2'
    ∧ branch (B @ [False]) T2' using Branching by blast
  then have branch (([[False] @ B] @ [True]) (Branching T1' T2')
    ∧ branch (([[False] @ B] @ [False]) (Branching T1' T2') by auto
    then have ?case by blast
}
ultimately show ?case using tree.exhaust by blast
qed

2.4 Internal Paths

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

lemma internal-inv-Leaf: ¬internal b Leaf using internal.simps by blast

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

lemma internal-inv-Branching-Right:
internal (Right#b) (Branching l r) ←→ internal b r
by auto
lemma internal-inv-Branching:
  \( \text{internal } p \ (\text{Branching } l \ r) \iff (p=\emptyset \lor (\exists a \ p'=a\#p' \land (a \rightarrow \text{internal } p' \ l) \land (\neg a \rightarrow \text{internal } p' \ r)) \ (\text{is } ?L \iff ?R) \)
proof
  assume \(?L\) then show \(?R\) by (metis internal.simps(2) neq-Nil-conv)
next
  assume \(r\) : \(?R\)
  then show \(?L\) proof
    assume \(p = \emptyset\) then show \(?L\) by auto
  next
    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:
  assumes \(\text{internal } \text{ds} \ T\)
  shows \(\text{path } \text{ds} \ T\)
using assms proof (induction \(T\) arbitrary: \(\text{ds}\))
  case Leaf
  then have False using internal-inv-Leaf by auto
  then show \(?case\) by auto
next
  case (\(\text{Branching } T_1 \ T_2\))
  then obtain \(a \ b\) where \(\text{ds-p: } \text{ds} = [\emptyset] \lor \text{ds} = a \# b \land (a \rightarrow \text{internal } b \ T_1) \land (\neg a \rightarrow \text{internal } b \ T_2)\) using internal-inv-Branching by blast
  then have \(\text{ds} = [\emptyset] \lor (a \rightarrow \text{path } b \ T_1) \land (\neg a \rightarrow \text{path } b \ T_2)\) using Branching by auto
  then show \(?case\) using \(\text{ds-p}\) by (cases a) auto
qed

lemma internal-prefix:
  assumes \(\text{internal } (\text{ds1} @ \text{ds2} @[d]) \ T\)
  shows \(\text{internal } \text{ds1} \ T\)
using assms proof (induction \(\text{ds1}\) arbitrary: \(T\))
  case (Cons a \(\text{ds1}\))
  then have \((\exists l \ r, \ T = \text{Branching } l \ r)\) using internal-inv-Leaf by (cases \(T\)) auto
  then obtain \(l \ r\) where \(\text{p-br: } T = \text{Branching } l \ r\) by auto
  show ?case proof (cases a)
    assume \(\text{atrue: a}\)
    then have \(\text{internal } ((\text{ds1} @ \text{ds2} @[d]) | l\) using \(\text{p-br Cons(2)}\) internal-inv-Branching by auto
    then have \(\text{internal } \text{ds1} \ l\) using \(\text{Cons(1)}\) by auto
    then show \(\text{internal } (a \# \text{ds1}) \ T\) using \(\text{p-br atrue}\) by auto
  next
next
assume \textit{afalse}: \sim a 
then have \textit{internal} \((\text{(ds1 @ ds2 @d) r using p-lr Cons(2)}\text{) internal-inv-Branching}\) 
by \textit{auto} 
then have \textit{internal ds1 r using Cons(1) by auto} 
then show \textit{internal} \((a \# \text{ds1}) \ T \text{ using p-lr afalse by auto}\) 
qed 
next 
case (Nil)  
then have \exists l r. T = \text{Branching l r using internal-inv-Leaf by (cases T) auto} 
then show ?case by \textit{auto} 
qed 

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

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

\textbf{2.5 Deleting Nodes} 

\textbf{fun delete ::} \textit{dir list} \Rightarrow \textit{tree} \Rightarrow \textit{tree where} 
delete \[] T = \text{Leaf} 
delete \((\text{True#ds}) \text{ (Branching T1 T2)} \Rightarrow \text{Branching (delete ds T1) T2}\)
\[
\text{delete } (\text{False} \# ds) (\text{Branching } T_1 T_2) = \text{Branching } T_1 (\text{delete } ds T_2)
\]
\[
\text{delete } (a \# ds) \text{ Leaf } = \text{Leaf}
\]

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

**lemma path-delete**: 
- **assumes** path \( p \) (delete \( ds \) \( T \))
- **shows** path \( p \) \( T \)
- **using** assms proof (induction \( p \) arbitrary: \( T \) \( ds \))
  - **case** Nil
  - then show ?case by simp
- **next**
  - **case** (Cons \( a \) \( p \))
  - then obtain \( b \) \( ds' \) where \( bds' \# p \)
    - \( ds = b \# ds' \) by (cases \( ds \)) auto
  - have \( \exists dT1 dT2. \text{delete } ds T = \text{Branching } dT1 dT2 \) using Cons path-inv-Cons by auto
  - then obtain \( dT1 dT2 \) where \( \text{delete } ds T = \text{Branching } dT1 dT2 \) by auto
- then have \( \exists T1 T2. T = \text{Branching } T1 T2 \)
  - by (cases \( T \); cases \( ds \)) auto
- then obtain \( T1 T2 \) where \( T1T2-p: T = \text{Branching } T1 T2 \) by auto

\[
\{
\begin{align*}
&\text{assume } a-p: a \\
&\text{assume } b-p: \neg b \\
&\text{have path } (a \# p) (\text{delete } ds T) \text{ using Cons by } - \\
&\quad \text{then have path } (a \# p) (\text{Branching } (T1) (\text{delete } ds' T2)) \text{ using } b-p bds'\# p \\
&\quad \text{using } T1T2-p \text{ by auto} \\
&\quad \text{then have path } p T1 \text{ using } a-p \text{ by auto} \\
&\quad \text{then have } ?\text{case using } T1T2-p a-p \text{ by auto}
\end{align*}
\}
\]
- **moreover**

\[
\{
\begin{align*}
&\text{assume } a-p: \neg a \\
&\text{assume } b-p: b \\
&\text{have path } (a \# p) (\text{delete } ds T) \text{ using Cons by } - \\
&\quad \text{then have path } (a \# p) (\text{Branching } (\text{delete } ds' T1) T2) \text{ using } b-p bds'\# p \\
&\quad \text{using } T1T2-p \text{ by auto} \\
&\quad \text{then have path } p T2 \text{ using } a-p \text{ by auto} \\
&\quad \text{then have } ?\text{case using } T1T2-p a-p \text{ by auto}
\end{align*}
\}
\]
- **moreover**

\[
\{
\begin{align*}
&\text{assume } a-p: a \\
&\text{assume } b-p: b \\
&\text{have path } (a \# p) (\text{delete } ds T) \text{ using Cons by } - \\
&\quad \text{then have path } (a \# p) (\text{Branching } (\text{delete } ds' T1) T2) \text{ using } b-p bds'\# p \\
&\quad \text{using } T1T2-p \text{ by auto}
\end{align*}
\}
\]

16
then have path p (delete ds' T1) using a-p by auto
then have path p T1 using Cons by auto
then have ?case using T1T2-p a-p by auto
}
moreover
{
  assume a-p: ¬a
  assume b-p: ¬b
  have path (a # p) (delete ds T) using Cons by –
  then have path (a # p) (Branching T1 (delete ds' T2)) using b-p bds'-p
  T1T2-p by auto
  then have path p (delete ds' T2) using a-p by auto
  then have path p T2 using Cons by auto
  then have ?case using T1T2-p a-p by auto
}
ultimately show ?case by blast
qed

lemma branch-delete:
  assumes branch p (delete ds T)
  shows branch p T ∨ p=ds
using assms proof (induction p arbitrary: T ds)
case Nil
  then have delete ds T = Leaf by (cases delete ds T) auto
  then have ds = [] ∨ T = Leaf using delete.elims by blast
  then show ?case by auto
next
case (Cons a p)
  then obtain b ds' where bds'-p: ds=b#ds' by (cases ds) auto
have ∃dT1 dT2. delete ds T = Branching dT1 dT2 using Cons path-inv-Cons
branch-is-path by blast
  then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto
  then have ∃T1 T2. T=Branching T1 T2
  by (cases T; cases ds) auto
  then obtain T1 T2 where T1T2-p: T=Branching T1 T2 by auto

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

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

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

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

then have ∃T1 T2. T=Branching T1 T2
  by (cases T; cases ds) auto
then obtain $T_1 \ T_2$ where $T_1T_2$-$p$: $T = \text{Branching } T_1 \ T_2$ by auto

\{
    \text{assume } a$-$p$: a \\
    \text{assume } b$-$p$: \neg b \\
    \text{then have } ?\text{case using } T_1T_2$-$p$ \ a$-$p$ \ b$-$p$ bds'$-p$ by auto
\}\n
moreover 
\{
    \text{assume } a$-$p$: \neg a \\
    \text{assume } b$-$p$: b \\
    \text{then have } ?\text{case using } T_1T_2$-$p$ \ a$-$p$ \ b$-$p$ bds'$-p$ by auto
\}\n
moreover 
\{
    \text{assume } a$-$p$: a \\
    \text{assume } b$-$p$: b \\
    \text{have path } (a \ # \ p) (\text{delete ds } T) \text{ using Cons by } -- \\
    \text{then have path } (a \ # \ p) (\text{Branching (delete ds' } T_1) \ T_2) \text{ using } b$-$p$ bds'$-p$
\}\n
$T1T2$-$p$ by auto \\
then have path $p$ (\text{delete ds' } T_1) \text{ using } a$-$p$ by auto \\
then have \text{\neg } (\exists c \ cs. \ p = ds' @ c \ # \ cs) \text{ using Cons by auto} \\
then have ?\text{case using } T1T2$-$p$ \ a$-$p$ \ b$-$p$ bds'$-p$ by auto
\}\n
moreover 
\{
    \text{assume } a$-$p$: \neg a \\
    \text{assume } b$-$p$: \neg b \\
    \text{have path } (a \ # \ p) (\text{delete ds } T) \text{ using Cons by } -- \\
    \text{then have path } (a \ # \ p) (\text{Branching } T_1 (\text{delete ds' } T_2)) \text{ using } b$-$p$ bds'$-p$ \ T1T2$-$p$ by auto \\
then have path $p$ (\text{delete ds' } T_2) \text{ using } a$-$p$ by auto \\
then have \text{\neg } (\exists c \ cs. \ p = ds' @ c \ # \ cs) \text{ using Cons by auto} \\
then have ?\text{case using } T1T2$-$p$ \ a$-$p$ \ b$-$p$ bds'$-p$ by auto
\}\n
ultimately show ?\text{case by blast}

qed

lemma \text{treetize-delete}: 
\text{assumes internal } p \ T \\
\text{shows treetize (delete } p \ T) < \text{treetize } T \\
\text{using assms proof (induction } p \ \text{arbitrary: } T) \\
\text{case (Nil)} \\
then have \exists T_1 \ T_2. \ T = \text{Branching } T_1 \ T_2 \text{ by (cases } T) \text{ auto} \\
then obtain T_1 \ T_2 \text{ where } T1T2$-$p$: $T = \text{Branching } T_1 \ T_2$ by auto \\
then show ?\text{case by auto}
\text{next} \\
\text{case (Cons a p)} \\
then have \exists T_1 \ T_2. \ T = \text{Branching } T_1 \ T_2 \text{ using path-inv-Cons internal-is-path}
by blast
then obtain \(T_1 \ T_2\) where \(T_{1T2-p}\): \(T = \text{Branching} \ (T_1 \ T_2)\) by auto
show \(?case\)
proof (cases \(a\))
assume \(a-p\): \(a\)
from \(a-p\) have \(\text{delete} \ (a\#p)\) \(T = (\text{Branching} \ (\text{delete} \ p \ T_1) \ T_2)\) using \(T_{1T2-p}\) by auto
moreover
from \(a-p\) have \(\text{internal} \ p \ T_1\) using \(T_{1T2-p}\) Cons by auto
then have \(\text{treesize} \ (\text{delete} \ p \ T_1) < \text{treesize} \ T_1\) using Cons by auto
ultimately
show \(?thesis\) using \(T_{1T2-p}\) by auto
next
assume \(a-p\): \(\neg a\)
from \(a-p\) have \(\text{delete} \ (a\#p)\) \(T = (\text{Branching} \ T_1 \ (\text{delete} \ p \ T_2))\) using \(T_{1T2-p}\) by auto
moreover
from \(a-p\) have \(\text{internal} \ p \ T_2\) using \(T_{1T2-p}\) Cons by auto
then have \(\text{treesize} \ (\text{delete} \ p \ T_2) < \text{treesize} \ T_2\) using Cons by auto
ultimately
show \(?thesis\) using \(T_{1T2-p}\) by auto
qed
qed

fun \(\text{cutoff}\) :: \((\text{dir list} \Rightarrow \text{bool}) \Rightarrow \text{dir list} \Rightarrow \text{tree} \Rightarrow \text{tree}\) where
\(\text{cutoff} \ \text{red} \ \text{ds} \ (\text{Branching} \ T_1 \ T_2) =\)
\(\text{if red} \ \text{ds} \ \text{then} \ \text{Leaf} \ \text{else} \ \text{Branching} \ (\text{cutoff} \ \text{red} \ (\text{ds}@[\text{Left}]) \ T_1) \ \text{cutoff} \ \text{red} \ (\text{ds}@[\text{Right}]) \ T_2)\)
| \(\text{cutoff} \ \text{red} \ \text{ds} \ \text{Leaf} = \ \text{Leaf}\)

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.

lemma \(\text{treesize-cutoff}\): \(\text{treesize} \ (\text{cutoff} \ \text{red} \ \text{ds} \ T) \leq \text{treesize} \ T\)
proof (induction \(T\) arbitrary: \(\text{ds}\))
\(\text{case Leaf then show } ?\text{case by auto}\)
next
\(\text{case (Branching} \ T_1 \ T_2)\)
\(\text{then have } \text{treesize} \ (\text{cutoff} \ \text{red} \ (\text{ds}@[\text{Left}]) \ T_1) + \text{treesize} \ (\text{cutoff} \ \text{red} \ (\text{ds}@[\text{Right}]) \ T_2) \leq \text{treesize} \ T_1 + \text{treesize} \ T_2\) using \(\text{add-mono}\) by blast
\(\text{then show } ?\text{case by auto}\)
qed

abbreviation \(\text{anypath}\) :: \(\text{tree} \Rightarrow (\text{dir list} \Rightarrow \text{bool}) \Rightarrow \text{bool}\) where
\(\text{anypath} \ T \ P \equiv \forall p. \ \text{path} \ p \ T \rightarrow P \ p\)

abbreviation \(\text{anybranch}\) :: \(\text{tree} \Rightarrow (\text{dir list} \Rightarrow \text{bool}) \Rightarrow \text{bool}\) where
\(\text{anybranch} \ T \ P \equiv \forall p. \ \text{branch} \ p \ T \rightarrow P \ p\)
abbreviation anyinternal :: tree ⇒ (dir list ⇒ bool) ⇒ bool where
  anyinternal T P ≡ ∀ p. internal p T ⇒ P p

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

21
from ds-p have \( \exists T = \text{Branching} \, ?T_1 \, ?T_2 \) by auto
from this b-p obtain \( a \, b' \, \text{where} \ b = a \, \# \, b' \land (a \rightarrow \text{branch} \, b' \, ?T_1') \land (\neg a \rightarrow \text{branch} \, b' \, ?T_2') \) using branch-inv-Branching[of \( b \, ?T_1' \, ?T_2' \)] by auto
then show \( \text{red}(ds@b) \) using aa bb by (cases a) auto
qed

then show ?case by blast
qed

lemma cutoff-branch:
assumes anybranch \( T \, (\lambda p. \, \text{red} \, p) \)
shows anybranch \( (\text{cutoff \, red} \, [] \, T) \, (\lambda p. \, \text{red} \, p) \)
using assms cutoff-branch[of \( T \, \text{red} \, [] \)] by auto

lemma cutoff-internal':
assumes anybranch \( T \, (\lambda b. \, \text{red}(ds@b)) \)
shows anyinternal \( (\text{cutoff \, red} \, ds \, T) \, (\lambda b. \, \neg \text{red}(ds@b)) \)
using assms proof (induction \( T \, \text{arbitrary:} \, ds \))
  case (Leaf) then show ?case using internal-inv-Leaf by simp
next
  case (Branching \( T_1 \, T_2 \))
  let \( ?T = \text{cutoff \, red} \, ds \, (\text{Branching} \, T_1 \, T_2) \)
  from Branching have \( \forall p. \, \text{branch} \, (\text{Left} \# p) \, (\text{Branching} \, T_1 \, T_2) \rightarrow \text{red} \, (ds \, @ \, (\text{Left} \# p)) \) by blast
  then have \( \forall p. \, \text{branch} \, p \, T_1 \rightarrow \text{red} \, (ds \, @ \, (\text{Left} \# p)) \) by auto
  then have anybranch \( T_1 \, (\lambda p. \, \text{red} \, ((ds \, @ \, [\text{Left}]) \, @ \, p)) \) by auto
  then have aa: anyinternal \( (\text{cutoff \, red} \, (ds \, @ \, [\text{Left}]) \, T_1) \, (\lambda p. \, \neg \, \text{red} \, ((ds \, @ \, [\text{Left}]) \, @ \, p)) \) using Branching by blast
  
  from Branching have \( \forall p. \, \text{branch} \, (\text{Right} \# p) \, (\text{Branching} \, T_1 \, T_2) \rightarrow \text{red} \, (ds \, @ \, (\text{Right} \# p)) \) by blast
  then have \( \forall p. \, \text{branch} \, p \, T_2 \rightarrow \text{red} \, (ds \, @ \, (\text{Right} \# p)) \) by auto
  then have anybranch \( T_2 \, (\lambda p. \, \text{red} \, ((ds \, @ \, [\text{Right}]) \, @ \, p)) \) by auto
  then have bb: anyinternal \( (\text{cutoff \, red} \, (ds \, @ \, [\text{Right}]) \, T_2) \, (\lambda p. \, \neg \, \text{red} \, ((ds \, @ \, [\text{Right}]) \, @ \, p)) \) using Branching by blast
\{
  fix \( p \)
  assume b-p: internal \( p \, ?T \)
  then have \( \neg \text{red} \, ds \) using internal-inv-Leaf by auto
  have \( p=\[] \lor p\#\[] \) by auto
  then have \( \neg \text{red}(ds@p) \)
  proof
  assume \( p=\[] \) then show \( \neg \text{red}(ds@p) \) using ds-p by auto
next
  let \( ?T_1' = \text{cutoff \, red} \, (ds@[\text{Left}]) \, T_1 \)
  let \( ?T_2' = \text{cutoff \, red} \, (ds@[\text{Right}]) \, T_2 \)
  assume \( p\#\[] \)
  moreover
  have \( ?T = \text{Branching} \, ?T_1' \, ?T_2' \) using ds-p by auto
22
ultimately obtain \( p' \) where b-p: \( p = a \# p' \land \)
\( (a \rightarrow \text{internal } p' \text{(cutoff red } ds @ \{\text{Left}\}) \ T_1) \land \)
\( (\neg a \rightarrow \text{internal } p' \text{(cutoff red } ds @ \{\text{Right}\}) \ T_2) \)
using b-p internal-ine-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:
assumes anybranch T red
shows anyinternal (cutoff red \[ T \]) \((\lambda p. \neg \text{red } p)\)
using assms cutoff-internal'[of T red \[] by auto

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

lemma cutoff-branch-internal:
assumes anybranch T red
shows \( \exists T'. \text{anyinternal } T'(\lambda p. \neg \text{red } p) \land \text{anybranch } T'(\lambda p. \text{red } p)\)
using assms cutoff-branch-internal' by blast

3 Possibly Infinite Trees

Possibly infinite trees are of type dir list set.

abbreviation wf-tree :: dir list set \( \Rightarrow \) bool where
\( \text{wf-tree } T \equiv (\forall ds d. (ds @ d) \in T \rightarrow ds \in T) \)
The subtree in with root r

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

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

lemma subtree-pos:
\( \text{subtree } T ds \subseteq \text{subtree } T (ds @ \{\text{Left}\}) \cup \text{subtree } T (ds @ \{\text{Right}\}) \cup \{ds\} \)
proof (rule subsetI; rule Set.UnCI)
let \( ?\text{subtree} = \text{subtree } T \)
fix x
assume asm: \( x \in ?\text{subtree } ds \)
assume \( x \notin \{ds\} \)
then have \( x \neq ds \) by simp
then have \( \exists e. d. x = ds \oplus [d] \oplus e \) using asm list.exhaust by auto
then have \( \exists e. x = ds \oplus [\text{Left}] \oplus e \lor \exists e. x = ds \oplus [\text{Right}] \oplus e \) using bool.exhaust by auto
then show \( x \in ?\text{subtree} (ds @ [\text{Left}]) \cup ?\text{subtree} (ds @ [\text{Right}]) \) using asm by auto
qed

3.1 Infinite Paths

abbreviation \( \text{wf-infpath} :: (\text{nat} \Rightarrow \\sigma \text{ list}) \Rightarrow \text{bool} \) where
\[ \text{wf-infpath } f \equiv (f \ 0 = []) \land (\forall n. \exists a. f (\text{Suc } n) = (f n) @ [a]) \]

lemma infpath-length:
assumes \( \text{wf-infpath } f \)
shows \( \text{length } (f n) = n \)
using assms proof (induction n)
case 0 then show ?case by auto
next
case (Suc n) then show ?case by (metis length-append-singleton)
qed

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

qed auto

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

lemma ith-in-extension:
assumes chain: \( \text{wf-infpath } f \)
assumes smalli: \( i < \text{length } (f n_1) \)
assumes \( n_1 n_2: n_1 \leq n_2 \)
shows \( f_{n_1} ! i = f_{n_2} ! i \)

**proof**

- from chain \( n_1 n_2 \) have \( \exists a. f_{n_1} @ a = f_{n_2} \) using chain-prefix by blast
- then obtain a where a-p: \( f_{n_1} @ a = f_{n_2} \) by auto
- have \( (f_{n_1} @ a) ! i = f_{n_1} ! i \) using smalli by (simp add: nth-append)
- then show ?thesis using a-p by auto

qed

### 4 König’s Lemma

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

**proof**

- let ?subtree = subtree T
- { assume asms: finite(?subtree(ds @ [Left]))
  finite(?subtree(ds @ [Right]))
  have ?subtree ds \subseteq ?subtree (ds @ [Left]) \cup ?subtree (ds @ [Right]) \cup \{ds\}
  using subtree-pos by auto
  then have finite(?subtree (ds)) using asms by (simp add: finite-subset)
  }
- then show \( \neg\text{finite}(\text{subtree } (\text{ds @ [Left]})) \lor \neg\text{finite}(\text{subtree } (\text{ds @ [Right]})) \)
using inf by auto

**qed**

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

where

- buildchain next 0 = []
- | buildchain next (Suc n) = next (buildchain next n)

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

**proof**

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

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

- from wellformed have prefix: \( \forall \text{ds } d. (\text{ds @ d}) \in T \longrightarrow \text{ds } \in T \) by blast

- { fix n
  have \( (?c n) \in T \land \neg\text{finite } (?\text{subtree } (?c n)) \)
  proof (induction n)
}

25
case 0
have \( \exists \, ds \in T \) using \( \text{inf} \) by (simp add: not-finite-existsD)
then obtain \( ds \) where \( ds \in T \) by auto
then have \( \{\emptyset \} \in T \) by auto
then have \( \emptyset \in T \) using \( \text{prefix} \) by blast
then show \( \text{?case} \) using \( \text{inf} \) by auto
next
  case \( \text{(Suc } n \) \)
  from \( \text{Suc} \) have \( \text{next-in: } \{\?c \, n\} \in T \) by auto
  from \( \text{Suc} \) have \( \text{next-inf: } \neg \text{finite } \{\?\text{subtree } \{\?c \, n\}\} \) by auto
  from \( \text{next-inf} \) have \( \text{next-next-inf: } \neg \text{finite } \{\?\text{subtree } \{\text{nextnode } \{\?c \, n\}\}\} \)
  using \( \text{inf-subs} \) by auto
  then have \( \exists \, ds \in \{\?\text{subtree } \{\text{nextnode } \{\?c \, n\}\}\} \) by (simp add: not-finite-existsD)
  then obtain \( ds \) where \( ds \in \{\?\text{subtree } \{\text{nextnode } \{\?c \, n\}\}\} \) by auto
  then have \( ds \in T \) \( \exists \, suf \in \{\?\text{subtree } \{\text{nextnode } \{\?c \, n\}\} \at \, suf \} \) by auto
  then obtain \( suf \) where \( suf \in T \land ds = (\{\text{nextnode } \{\?c \, n\}\} \at \, suf \) by auto
  then have \( \{\text{nextnode } \{\?c \, n\}\} \in T \)
  using \( \text{prefix} \) by blast
  then have \( \{\, \varnothing \, (\text{Suc } n)\} \in T \) by auto
  then show \( \text{?case} \) using \( \text{next-next-inf} \) by auto
  qed

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

5 More Terms and Literals

theory \( \text{Resolution} \) imports \( \text{TermsAndLiterals} \) \( \text{Tree} \) begin

fun \( \text{complement} :: \, \text{'}t \) literal \( \Rightarrow \, \text{'}t \) literal \( \cdot \) where
\( \cdot \text{Pos } P \) ts\(\cdot\) = \( \text{Neg } P \) ts
\( \cdot \text{Neg } P \) ts\(\cdot\) = \( \text{Pos } P \) ts

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

lemma \( \text{cancel-comp2} : \)\(\text{asm: } l_1^c = l_2^c \)
  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 \( l_1 = (l_2^c)^c \) using \( \text{cancel-comp1[of } l_1 \) by auto
  then show \( \text{thesis} \) using \( \text{cancel-comp1[of } l_2 \) by auto
  qed

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

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

6 Clauses

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

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

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

lemma cancel-compls2: \( \text{assumes asm: } L_1^C = L_2^C \)
  \( \text{shows } L_1 = L_2 \)
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 :: \( 't \text{ term } \Rightarrow \text{var-sym set} \)
where
\( \text{vars}_1 \ (\text{Var } x) = \{ x \} \)
| \( \text{vars}_1 \ (\text{Fun } f ts) = (\bigcup t \in \text{set } ts. \text{vars}_1 \ t) \)
abbreviation \( \text{vars}_{ts} \) :: \( \text{fterm list} \Rightarrow \text{var-sym set} \)
\[ \text{vars}_{ts} \equiv (\bigcup t \in \text{set ts. vars}_t) \]
definition \( \text{vars}_l \) :: \( \text{fterm literal} \Rightarrow \text{var-sym set} \)
\[ \text{vars}_l l = \text{vars}_{ts} (\text{get-terms } l) \]
definition \( \text{vars}_{ls} \) :: \( \text{fterm literal set} \Rightarrow \text{var-sym set} \)
\[ \text{vars}_{ls} L \equiv \bigcup l \in L. \text{vars}_l l \]

lemma \( \text{ground-vars}_t \):
\begin{align*}
\text{assumes} & \text{ground } t \\
\text{shows} & \text{vars}_t t = {} \\
\end{align*}
using \( \text{assms} \) by (induction \( t \)) auto

lemma \( \text{ground-vars}_{ts} \):
\begin{align*}
\text{assumes} & \text{ground}_{ts} ts \\
\text{shows} & \text{vars}_{ts} ts = {} \\
\end{align*}
using \( \text{assms} \) \( \text{ground-vars}_t \) by auto

lemma \( \text{ground-vars}_{ls} \):
\begin{align*}
\text{assumes} & \text{ground}_{ls} L \\
\text{shows} & \text{vars}_{ls} L = {} \\
\end{align*}
unfolding \( \text{vars}_{ls}-\text{def} \) using \( \text{assms} \) \( \text{ground-vars}_t \) by auto

lemma \( \text{ground-comp} \):
\[ \text{ground}_i (l^c) \iff \text{ground}_i l \]
by \( \text{(cases } l \) auto

lemma \( \text{ground-compls} \):
\[ \text{ground}_{ls} (L^c) \iff \text{ground}_{ls} L \]
using \( \text{ground-comp} \) by auto

7 Semantics

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

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

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

fun \( \text{eval}_l \) :: \( 'u \text{ var-denot} \Rightarrow 'u \text{ fun-denot} \Rightarrow 'u \text{ pred-denot} \Rightarrow \text{fterm literal} \Rightarrow \text{bool} \)
\[ \text{eval}_l E F G (\text{Pos } p ts) \iff G p (\text{eval}_{ls} E F ts) \]
\begin{verbatim}
| eval \_ E F G (Neg p ts) \hbox{\lra} \neg G \; (eval\_ts E F ts)

definition eval\_c :: 'a fun-denot \Rightarrow 'a pred-denot \Rightarrow fterm clause \Rightarrow bool where
  eval\_c \; F G C \hbox{\lra} (\forall E. \exists l \in C. eval\_l E F G l)

definition eval\_cs :: 'a fun-denot \Rightarrow 'a pred-denot \Rightarrow fterm clause set \Rightarrow bool where
  eval\_cs F G Cs \hbox{\lra} (\forall C \in Cs. eval\_c F G C)

7.1 Semantics of Ground Terms

lemma ground-var-denott:
  assumes ground\_t t
  shows eval\_t E F t = eval\_t E' F t
  using assms proof (induction t)
    case (Var x)
      then have False by auto
      then show ?case by auto
    next
    case (Fun f ts)
      then have \\!

lemma ground-var-denotts:
  assumes ground\_ts ts
  shows eval\_ts E F ts = eval\_ts E' F ts
  using assms ground-var-denott by (metis map-eq-conv)

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

8 Substitutions

type-synonym substitution = var-sym \Rightarrow fterm

fun sub :: fterm \Rightarrow substitution \Rightarrow fterm (infixl \cdot 55) where
\end{verbatim}
\[(\text{Var } x) \cdot \iota \sigma = \sigma x\]

| \(\text{Fun } f\ ts\) \cdot \iota \sigma = \text{Fun } f\ (\text{map } (\lambda t.\ t \cdot \iota \sigma)\ ts)\)

**abbreviation** \(\text{subs} :: \text{fterm list} \Rightarrow \text{substitution} \Rightarrow \text{fterm list} \) (infixl \(\iota_s\) 55) where

\[\text{ts} \iota_s \sigma \equiv (\text{map } (\lambda t.\ t \cdot \iota \sigma)\ ts)\]

**fun** \(\text{subl} :: \text{fterm literal} \Rightarrow \text{substitution} \Rightarrow \text{fterm literal} \) (infixl \(\cdot\) 55) where

\[(\text{Pos } p\ ts) \cdot \iota \sigma = \text{Pos } p\ (\text{ts} \iota_s \sigma)\]

\[(\text{Neg } p\ ts) \cdot \iota \sigma = \text{Neg } p\ (\text{ts} \iota_s \sigma)\]

**abbreviation** \(\text{subls} :: \text{fterm literal set} \Rightarrow \text{substitution} \Rightarrow \text{fterm literal set} \) (infixl \(\cdot\) 55) where

\[\text{L} \cdot \iota_s \sigma \equiv (\lambda l.\ l \cdot \iota \sigma)\ ' L\]

**lemma** \(\text{subls-def2}: \text{L} \cdot \iota_s \sigma = \{l \cdot \iota \sigma | l.\ l \in L\}\) by auto

**definition** \(\text{instance-of}_1 :: \text{fterm} \Rightarrow \text{fterm} \Rightarrow \text{bool} \) where

\[\text{instance-of}_1 t_1 t_2 \leftarrow (\exists \sigma.\ t_1 = t_2 \cdot \iota \sigma)\]

**definition** \(\text{instance-of}_s :: \text{fterm list} \Rightarrow \text{fterm list} \Rightarrow \text{bool} \) where

\[\text{instance-of}_s ts_1 ts_2 \leftarrow (\exists \sigma.\ ts_1 = ts_2 \cdot \iota_s \sigma)\]

**definition** \(\text{instance-of}_1 :: \text{fterm literal} \Rightarrow \text{fterm literal} \Rightarrow \text{bool} \) where

\[\text{instance-of}_1 l_1 l_2 \leftarrow (\exists \sigma.\ l_1 = l_2 \cdot \iota \sigma)\]

**definition** \(\text{instance-of}_s :: \text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{bool} \) where

\[\text{instance-of}_s C_1 C_2 \leftarrow (\exists \sigma.\ C_1 = C_2 \cdot \iota_s \sigma)\]

**lemma** \(\text{comp-sub}: (l^c) \cdot \iota \sigma = (l \cdot \iota \sigma)^c\)

by (cases l) auto

**lemma** \(\text{compls-subls}: (L^C) \cdot \iota_s \sigma = (L \cdot \iota_s \sigma)^C\)

using \(\text{comp-sub}\) apply auto

apply (metis image-eqI)

done

**lemma** \(\text{subls-union}: (L_1 \cup L_2) \cdot \iota_s \sigma = (L_1 \cdot \iota_s \sigma) \cup (L_2 \cdot \iota_s \sigma)\) by auto

**definition** \(\text{var-renaming-of} :: \text{fterm clause} \Rightarrow \text{fterm clause} \Rightarrow \text{bool} \) where

\[\text{var-renaming-of } C_1 C_2 \leftarrow \text{instance-of}_s C_1 C_2 \land \text{instance-of}_s C_2 C_1\]

### 8.1 The Empty Substitution

**abbreviation** \(\varepsilon :: \text{substitution} \) where

\[\varepsilon \equiv \text{Var}\]

**lemma** \(\text{empty-subt}: (t :: \text{fterm}) \cdot \varepsilon = t\)

by (induction t) (auto simp add: map-idI)
lemma empty-subts: ts \cdot ts \in = ts
using empty-subt by auto

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

lemma empty-subls: L \cdot Ls \in = L
using empty-subl by auto

lemma instance-of_{ts-self}: instance-of_{ts} ts ts
unfolding instance-of_{ts-def}
proof
  show ts = ts \cdot ts \in using empty-subts by auto
qed

lemma instance-of_{ts-self}: instance-of_{ts} ts ts
unfolding instance-of_{ts-def}
proof
  show ts = ts \cdot ts \in using empty-subts by auto
qed

lemma instance-of_{l-self}: instance-of_{l} l l
unfolding instance-of_{l-def}
proof
  show l = l \cdot l \in using empty-subl by auto
qed

lemma instance-of_{ts-self}: instance-of_{ts} L L
unfolding instance-of_{ts-def}
proof
  show L = L \cdot Ls \in using empty-subls by auto
qed

8.2 Substitutions and Ground Terms

lemma ground-sub:
  assumes ground_{t} t
  shows t \cdot, \sigma = t
using assms by (induction t) (auto simp add: map-idI)

lemma ground-subs:
  assumes ground_{ts} ts
  shows ts \cdot ts \sigma = ts
using assms ground-sub by (simp add: map-idI)

lemma ground_{l-sub}:
  assumes ground_{l} l
  shows l \cdot l \sigma = l
using assms ground-subs by (cases l) auto

lemma ground1s-subs:
  assumes ground: ground1s L
  shows \( L \cdot \sigma = L \)
proof -
\{
  fix l
  assume l-L: \( l \in L \)
  then have ground \( l \) using ground by auto
  then have \( l = l \cdot \sigma \) using ground1-subs by auto
  moreover
  then have \( l \cdot \sigma \in L \cdot \sigma \) using l-L by auto
  ultimately
  have \( l \in L \cdot \sigma \) by auto
\}
moreover
\{
  fix l
  assume l-\( \cdot \sigma \): \( l \in L \cdot \sigma \)
  then obtain \( l' \) where \( l' \cdot \cdot \sigma \) \( l' \in L \land l' \cdot \sigma = l \) by auto
  then have \( l' = l \) using ground ground1-subs by auto
  from l-L l-\( \cdot \sigma \) 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
\( (\sigma_1 \cdot \sigma_2) \cdot x = (\sigma_1 \cdot x) \cdot \sigma_2 \)

lemma composition-conseq2t: \( (t \cdot \sigma_1) \cdot \cdot \sigma_2 = t \cdot (\sigma_1 \cdot \sigma_2) \)
proof (induction \( t \))
case \( \text{Var} \ x \)
  have \( ((\text{Var} \ x) \cdot \cdot \sigma_1) \cdot \cdot \sigma_2 = (\sigma_1 \cdot x) \cdot \cdot \sigma_2 \) by simp
  also have \( \ldots = (\sigma_1 \cdot \sigma_2) \cdot x \) unfolding composition-def by simp
  finally show \( \text{case} \) by auto
next
case \( \text{Fun} \ t \ t s \)
  then show \( \text{case unfolding composition-def} \) by auto
qed

lemma composition-conseq2ts: \( t s \cdot t s \cdot \sigma_1 \cdot \sigma_2 = t s \cdot \cdot (\sigma_1 \cdot \sigma_2) \)
using composition-conseq2t by auto

lemma composition-conseq2l: \( l \cdot \sigma_1 \cdot \cdot \sigma_2 = l \cdot (\sigma_1 \cdot \sigma_2) \)
using composition-conseq2t by (cases l) auto
lemma composition-conseq2ls: \((L \cdot \sigma_1) \cdot \sigma_2 = L \cdot (\sigma_1 \cdot \sigma_2)\)
using composition-conseq2l apply auto
apply (metis imageI)
done

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

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

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

lemma instance-of-t-trans :
  assumes \(t_{12} : \text{instance-of}_t \ t_1 \ t_2\)
  assumes \(t_{23} : \text{instance-of}_t \ t_2 \ t_3\)
  shows \(\text{instance-of}_t \ t_1 \ t_3\)
proof
  from \(t_{12}\) obtain \(\sigma_{12}\) where \(t_1 = t_2 \cdot \sigma_{12}\)
  unfolding \(\text{instance-of}_t\)-def by auto
  moreover
  from \(t_{23}\) obtain \(\sigma_{23}\) where \(t_2 = t_3 \cdot \sigma_{23}\)
  unfolding \(\text{instance-of}_t\)-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 \(?thesis\)
  unfolding \(\text{instance-of}_t\)-def by auto
qed

lemma instance-of-ts-trans :
  assumes \(ts_{12} : \text{instance-of}_{ts} \ ts_1 \ ts_2\)
  assumes \(ts_{23} : \text{instance-of}_{ts} \ ts_2 \ ts_3\)
  shows \(\text{instance-of}_{ts} \ ts_1 \ ts_3\)
proof
  from \(ts_{12}\) obtain \(\sigma_{12}\) where \(ts_1 = ts_2 \cdot ts_1 \cdot \sigma_{12}\)
  unfolding \(\text{instance-of}_{ts}\)-def by auto
moreover
from $ts_{23}$ obtain $\sigma_{23}$ where $ts_{2} = ts_{23} \cdot ts_{23}

unfolding instance-of$_{ts}$-def by auto
ultimately
have $ts_{1} = (ts_{3} \cdot ts_{23}) \cdot ts_{12}$ by auto
then have $ts_{1} = ts_{3} \cdot (ts_{23} \cdot ts_{12})$ using composition-conseq2ts by simp
then show ?thesis unfolding instance-of$_{ts}$-def by auto
qed

lemma instance-of$_{l}$-trans :
  assumes $l_{12}$: instance-of$_{l}$ $l_{1}$ $l_{2}$
  assumes $l_{23}$: instance-of$_{l}$ $l_{2}$ $l_{3}$
  shows instance-of$_{l}$ $l_{1}$ $l_{3}$
proof –
  from $l_{12}$ obtain $\sigma_{12}$ where $l_{1} = l_{2} \cdot l_{12}$
    unfolding instance-of$_{l}$-def by auto
moreover
  from $l_{23}$ obtain $\sigma_{23}$ where $l_{2} = l_{3} \cdot l_{23}$
    unfolding instance-of$_{l}$-def by auto
ultimately
  have $l_{1} = (l_{3} \cdot l_{23}) \cdot l_{12}$ by auto
  then have $l_{1} = l_{3} \cdot (l_{23} \cdot l_{12})$ using composition-conseq2l by simp
then show ?thesis unfolding instance-of$_{l}$-def by auto
qed

lemma instance-of$_{ls}$-trans :
  assumes $L_{12}$: instance-of$_{ls}$ $L_{1}$ $L_{2}$
  assumes $L_{23}$: instance-of$_{ls}$ $L_{2}$ $L_{3}$
  shows instance-of$_{ls}$ $L_{1}$ $L_{3}$
proof –
  from $L_{12}$ obtain $\sigma_{12}$ where $L_{1} = L_{2} \cdot l_{12}$
    unfolding instance-of$_{ls}$-def by auto
moreover
  from $L_{23}$ obtain $\sigma_{23}$ where $L_{2} = L_{3} \cdot l_{23}$
    unfolding instance-of$_{ls}$-def by auto
ultimately
  have $L_{1} = (L_{3} \cdot L_{23}) \cdot l_{12}$ by auto
  then have $L_{1} = L_{3} \cdot L_{23} \cdot l_{12}$ using composition-conseq2ls by simp
then show ?thesis unfolding instance-of$_{ls}$-def by auto
qed

8.4 Merging substitutions

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

lemma relevant-vars-subt:
  assumes \( \forall x \in \text{vars}_t. t. \sigma_1 x = \sigma_2 x \)
  shows \( \forall t. \sigma_1 = t \gamma \sigma_2 \)
using assms proof (induction \( t \))
  case (Fun \( f ts \))
  have \( \forall t. \in \text{set} ts \rightarrow \text{vars}_t t \subseteq \text{vars}_t ts \) by (induction \( ts \)) auto
  have \( \forall \in \text{set} ts. t \gamma t. \sigma_1 = t \gamma t. \sigma_2 \)
  proof
    fix \( t \)
    assume \( \in \text{set} ts \)
    then have \( \forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x \) using \( f \) Fun(2) by auto
    then show \( \forall t. \gamma t. \sigma_1 = t \gamma t. \sigma_2 \) using Fun \( t \text{ints} \) by auto
qed
  then have \( \gamma t. \sigma_1 = t \gamma t. \sigma_2 \) by auto
  then show \(?case\) by auto
qed auto

lemma relevant-vars-subts:
  assumes \( \forall x \in \text{vars}_t ts. \sigma_1 x = \sigma_2 x \)
  shows \( \forall \in \text{set} ts. \gamma t. \sigma_1 = t \gamma t. \sigma_2 \)
proof –
  have \( \forall t. \in \text{set} ts \rightarrow \text{vars}_t t \subseteq \text{vars}_t ts \) by (induction \( ts \)) auto
  have \( \forall \in \text{set} ts. t \gamma \sigma_1 = t \gamma \sigma_2 \)
  proof
    fix \( t \)
    assume \( \in \text{set} ts \)
    then have \( \forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x \) using \( f \) asm by auto
    then show \( \forall t. \gamma t. \sigma_1 = t \gamma t. \sigma_2 \) using relevant-vars-subt \( t \text{ints} \) by auto
  qed
  then show \(?thesis\) by auto
qed

35
lemma relevant-vars-subl:
  assumes \( \forall x \in \text{vars}_l. \sigma_1 x = \sigma_2 x \)
  shows \( l \cdot \sigma_1 = l \cdot \sigma_2 \)
  using \text{assms} \ proof \ (\text{induction } l) \n  \begin{aligned} \text{case (Pos } p\text{ ts)} \\ & \text{then show } \ ? \text{ case using relevant-vars-subts unfolding vars}_l\text{-def by auto} \end{aligned} 
  \begin{aligned} \text{next} \\ \text{case (Neg } p\text{ ts)} \\ & \text{then show } \ ? \text{ case using relevant-vars-subts unfolding vars}_l\text{-def by auto} \end{aligned} 
  \text{qed}

lemma relevant-vars-subls:
  assumes \text{asm: } \forall x \in \text{vars}_{ls} L. \sigma_1 x = \sigma_2 x \n  shows \( L \cdot \text{ls } \sigma_1 = L \cdot \text{ls } \sigma_2 \)
  \text{proof} −
  \begin{aligned} \text{have } f: \forall l, l \in L \rightarrow \text{vars}_l l \subseteq \text{vars}_{ls} L \text{ unfolding vars}_{ls}-\text{def by auto} \quad & \text{have } \forall l \in L. l \cdot \sigma_1 = l \cdot \sigma_2 \text{ proof −} \end{aligned} 
  \begin{aligned} \text{fix } l \\ \text{assume } \text{linls: } l \in L \\ \text{then have } \forall x \in \text{vars}_l l. \sigma_1 x = \sigma_2 x \text{ using } f \text{ asm by auto} \quad & \text{then show } l \cdot \sigma_1 = l \cdot \sigma_2 \text{ using relevant-vars-subl linls by auto} \end{aligned} 
  \text{then show } ? \text{ thesis by (meson image-cong)} 
  \text{qed}

lemma merge-sub:
  assumes \text{dist: } \forall x \in \text{vars}_{ls} C \cap \text{vars}_{ls} D = \{\} 
  \text{assumes } CC': C \cdot_{ls} \text{ lmbd } = C' 
  \text{assumes } DD': D \cdot_{ls} \mu = D' 
  shows \exists \eta. C \cdot_{ls} \eta = C' \land D \cdot_{ls} \eta = D' 
  \text{proof} −
  \begin{aligned} \text{let } ?\eta = \lambda x. \text{ if } x \in \text{vars}_{ls} C \then \text{ lmbd } x \text{ else } \mu x \\ \text{have } \forall x \in \text{vars}_{ls} C. ?\eta x = \text{ lmbd } x \text{ by auto} \\ \text{then have } C \cdot_{ls} ?\eta = C' \text{ using relevant-vars-subls[of } C ?\eta \text{ lmbd] by auto} \\
\text{then have } C \cdot_{ls} ?\eta = C' \text{ using } CC' \text{ by auto} \end{aligned} 
  \begin{aligned} \text{moreover} \\ \text{have } \forall x \in \text{vars}_{ls} D, ?\eta x = \mu x \text{ using dist by auto} \\ \text{then have } D \cdot_{ls} ?\eta = D' \text{ using relevant-vars-subls[of } D ?\eta \mu] \text{ by auto} \\
\text{then have } D \cdot_{ls} ?\eta = D' \text{ using } DD' \text{ by auto} \end{aligned} 
  \text{ultimately} 
  \text{show } ?\text{thesis by auto} 
  \text{qed}

8.5 Standardizing apart

abbreviation std1 :: \text{fterm clause } \Rightarrow \text{fterm clause where}\n
std₁ C ≡ C ⋙₁s (λx. Var ("1" @ x))

abbreviation std₂ :: fterm clause ⇒ fterm clause where
std₂ C ≡ C ⋙₂s (λx. Var ("2" @ x))

lemma std-apart-apart':
  assumes x ∈ vars₁ t (λ₁ (λx:char list. Var (y @ x)))
  shows ∃x'. x = y@x'
  using assms by (induction t) auto

lemma std-apart-apart'':
  assumes x ∈ vars₁ t (λ₁ (λx:char list. Var (y@x)))
  shows ∃x'. x = y@x'
  using assms unfolding vars₁-def using std-apart-apart'' by (cases l) auto

lemma std-apart-apart:
  vars ls (std₁ C₁) ∩ vars ls (std₂ C₂) = {}
  proof −
  { fix x
    assume xin: x ∈ vars ls (std₁ C₁) ∩ vars ls (std₂ C₂)
    from xin have x ∈ vars ls (std₁ C₁) by auto
    then have ∃x'. x = "1" @ x'
      using std-apart-apart' [of x - "1"] unfolding vars₁-def by auto
    moreover
    from xin have x ∈ vars ls (std₂ C₂) by auto
    then have ∃x'. x = "2" @ x'
      using std-apart-apart' [of x - "2"] unfolding vars₁-def by auto
    ultimately have False by auto
    then have x ∈ {} by auto
  }
  then show ?thesis by auto
qed

lemma std-apart-instance-of₁s₁: instance-of₁s₁ C₁ (std₁ C₁)
  proof −
  have empty: (λx. Var ("1"@x)) ⋙₁s (λx. Var (tl x)) = ε using composition-def by auto
  have C₁ ⋙₁s ε = C₁ using empty-subls by auto
  then have C₁ ⋙₁s ((λx. Var ("1"@x)) ⋙₁s (λx. Var (tl x))) = C₁ using empty by auto
  then have (C₁ ⋙₁s (λx. Var ("1"@x))) ⋙₁s (λx. Var (tl x)) = C₁ using composition-conseq2ls by auto
  then have C₁ = (std₁ C₁) ⋙₁s (λx. Var (tl x)) by auto
  then show instance-of₁s₁ C₁ (std₁ C₁) unfolding instance-of₁s₁-def by auto
qed

lemma std-apart-instance-of₁s₂: instance-of₁s₁ C₂ (std₂ C₂)
  proof −
have empty: \((\lambda x. \text{Var} ("^2\text{@x}x)) \cdot (\lambda x. \text{Var} (tl x)) = \varepsilon\) using composition-def
by auto

have \(C_2 \cdot_1 \varepsilon = C_2\) using empty-subls by auto
then have \(C_2 \cdot_1 ((\lambda x. \text{Var} ("^2\text{@x}x)\cdot (\lambda x. \text{Var} (tl x))) = C_2\) using empty
by auto
then have \(C_2 = (std_2 C_2) \cdot_1 (\lambda x. \text{Var} (tl x))\) by auto
then show instance-of \(C_2\) \(C_2 (std_2 C_2)\) unfolding instance-of-\(ts\)-def by auto
qed

9 Unifiers

definition unifier_1ts :: substitution \(\Rightarrow\) fterm set \(\Rightarrow\) bool where
unifier_1ts \(\sigma\) \(ts\) \(\longleftrightarrow\) \((\exists t'. \forall t \in ts. t \cdot_1 \sigma = t')\)
definition unifier_1ts :: substitution \(\Rightarrow\) fterm literal set \(\Rightarrow\) bool where
unifier_1ts \(\sigma\) \(L\) \(\longleftrightarrow\) \((\exists l'. \forall l \in L. l \cdot_1 \sigma = l')\)

lemma unif-sub:
assumes unif: unifier_1ts \(\sigma\) \(L\)
assumes nonempty: \(L \neq \{\}\)
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 \(L \cdot_1 \sigma = \{l \cdot_1 \sigma\}\) unfolding unifier_1ts-def by auto
  then show \(?\)thesis by auto
 qed

lemma unifier-t-def2:
assumes \(L\)-elem: \(ts \neq \{\}\)
shows unifier_1ts \(\sigma\) \(ts\) \(\longleftrightarrow\) \((\exists l. (\lambda t. \text{sub}\ t\ \sigma) \cdot\ ts = \{l\})\)
proof
  assume unif: unifier_1ts \(\sigma\) \(ts\)
  from \(L\)-elem obtain \(t\) where \(t \in ts\) by auto
  then have \((\lambda t. \text{sub}\ t\ \sigma) \cdot\ ts = \{t \cdot_1 \sigma\}\) using unif unfolding unifier_1ts-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 \(\forall l' \in ts. l' \cdot_1 \sigma = l\) by auto
  then show unifier_1ts \(\sigma\) \(ts\) unfolding unifier_1ts-def by auto
qed

lemma unifier_1ts-def2:
assumes \(L\)-elem: \(L \neq \{\}\)
shows unifier_1ts \(\sigma\) \(L\) \(\longleftrightarrow\) \((\exists l. L \cdot_1 \sigma = \{l\})\)
proof
  assume unif: unifier₁ₛ σ L
  from L-elem obtain l where l ∈ L by auto
  then have L ·ₜₛ σ = {l} using unif unfolding unifier₁ₛ-def by auto
  then show ∃ l. L ·ₜₛ σ = {l} by auto
next
  assume ∃ l. L ·ₜₛ σ = {l}
  then obtain l where L ·ₜₛ σ = {l} by auto
  then have ∀ l’ ∈ L. l’ ·ₜₛ σ = l by auto
  then show unifier₁ₛ σ L unfolding unifier₁ₛ-def by auto
qed

lemma ground₁ₛ-unif-singleton:
  assumes ground₁ₛ: ground₁ₛ L
  assumes unif: unifier₁ₛ σ’ L
  assumes empt: L ≠ {} shows ∃ l. L = {l}
proof –
  from unif empt have ∃ l. L ·ₜₛ σ’ = {l} using unif-sub by auto
  then show thesis using ground₁ₛ-subls ground₁ₛ by auto
qed

definition unifiablets :: fterm set ⇒ bool where
  unifiablets fs ←→ (∃ σ. unifier ts σ fs)

definition unifiablels :: fterm literal set ⇒ bool where
  unifiablels L ←→ (∃ σ. unifier ls σ L)

lemma unifier-comp[simp]: unifier₁ₛ σ (Lᶜ) ←→ unifier₁ₛ σ L
proof
  assume unifier₁ₛ σ (Lᶜ)
  then obtain l’ where l’-p: ∀ l ∈ Lᶜ. l ·ₜ l σ = l’
  unfolding unifier₁ₛ-def by auto
  obtain l’ where (l’)^c = l’ using comp-exi2[of l’] by auto
  from this l’-p have l’-p: ∀ l ∈ Lᶜ. l ·ₜ l σ = (l’)^c by auto
  have ∀ l ∈ L. l ·ₜ l σ = l’
  proof
    fix l
    assume l ∈ L
    then have l’ ∈ Lᶜ by auto
    then have (l’)^c ·ₜ l σ = (l’)^c using l’-p by auto
    then have (l )^c ·ₜ l σ = (l)^c by (cases l) auto
    then show l ·ₜ l σ = l’ using cancel-comp2 by blast
  qed
  then show unifier₁ₛ σ L unfolding unifier₁ₛ-def by auto
next
  assume unifier₁ₛ σ L
  then obtain l’ where l’-p: ∀ l ∈ L. l ·ₜ l σ = l’ unfolding unifier₁ₛ-def by auto
  have ∀ l ∈ Lᶜ. l ·ₜ l σ = (l’)^c
39
proof
  fix l
  assume \( l \in L \)
  then have \( l \in L \) using cancel-comp1 by (metis image-iff)
  then show \( \sigma(l) = (l) \in C \)
  using l-p comp-sub cancel-comp1 by metis
  then show unifier_{ts} \( (L) \) unfolding unifier_{ts}-def by auto
qed

lemma unifier-sub1:
  assumes unifier_{ts} \( (L) \)
  assumes \( L' \subseteq L \)
  shows unifier_{ts} \( (L') \)
  using assms unfolding unifier_{ts}-def by auto

lemma unifier-sub2:
  assumes asm: unifier_{ts} \( (L_1 \cup L_2) \)
  shows unifier_{ts} \( L_1 \land \) unifier_{ts} \( L_2 \)
  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 mgu_{ts} :: substitution \( \Rightarrow \) fterm set \( \Rightarrow \) bool where
  \( mgu_{ts} \sigma \text{ ts} \leftrightarrow \text{unifier}_{ts} \sigma \text{ ts} \land (\forall u. \text{unifier}_{ts} u \text{ ts} \rightarrow (\exists i. u = \sigma \cdot i)) \)

definition mgu_{tls} :: substitution \( \Rightarrow \) fterm literal set \( \Rightarrow \) bool where
  \( mgu_{ts} \sigma \text{ L} \leftrightarrow \text{unifier}_{ts} \sigma \text{ L} \land (\forall u. \text{unifier}_{ts} u \text{ L} \rightarrow (\exists i. u = \sigma \cdot i)) \)

10 Resolution

definition applicable :: fterm clause \( \Rightarrow \) fterm clause
  \( \Rightarrow \) fterm literal set \( \Rightarrow \) fterm literal set
  \( \Rightarrow \) substitution \( \Rightarrow \) bool where
  applicable \( C_1 \) \( C_2 \) \( L_1 \) \( L_2 \) \( \sigma \leftrightarrow \)
  \( \forall \text{ vars}_{ts} \text{ par} \cdot (C_1 \setminus \text{ vars}_{ts} \text{ C}_2 \subseteq \{\}) \land (L_1 \subseteq \{\}) \land (L_2 \subseteq \{\}) \land (\forall u. \text{unifier}_{ts} u \text{ L} \rightarrow (\exists i. u = \sigma \cdot i)) \)

declaration mresolution :: fterm clause \( \Rightarrow \) fterm clause
  \( \Rightarrow \) fterm literal set \( \Rightarrow \) fterm literal set
  \( \Rightarrow \) substitution \( \Rightarrow \) fterm clause where
  mresolution \( C_1 \) \( C_2 \) \( L_1 \) \( L_2 \) \( \sigma \leftrightarrow \)
  \( \forall \text{ vars}_{ts} \text{ par} \cdot (C_1 \setminus \text{ vars}_{ts} \text{ C}_2 \subseteq \{\}) \land (L_1 \subseteq \{\}) \land (L_2 \subseteq \{\}) \land (\forall u. \text{unifier}_{ts} u \text{ L} \rightarrow (\exists i. u = \sigma \cdot i)) \)

declaration resolution :: fterm clause \( \Rightarrow \) fterm clause
\[ ⇒ \text{term literal set} \Rightarrow \text{term literal set} \]
\[ ⇒ \text{substitution} \Rightarrow \text{term clause where} \]
 resolution \( C_1 \ C_2 \ L_1 \ L_2 \ σ = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot \ls \ σ \]

**inductive mresolution-step** :: \text{term clause set} ⇒ \text{term clause set} ⇒ \text{bool where}

mresolution-rule:
- \( C_1 \in Cs \implies C_2 \in Cs \implies \text{applicable} \ C_1 \ C_2 \ L_1 \ L_2 \ σ \implies \text{mresolution-step} \ Cs \ (Cs \cup \{\text{mresolution} \ C_1 \ C_2 \ L_1 \ L_2 \ σ\}) \]
- standardize-apart:
  - \( C \in Cs \implies \text{var-renaming-of} \ C \ C' \implies \text{mresolution-step} \ Cs \ (Cs \cup \{C'\}) \)

**definition mresolution-deriv** :: \text{term clause set} ⇒ \text{term clause set} ⇒ \text{bool where}

mresolution-deriv = rtranclp mresolution-step

**definition resolution-deriv** :: \text{term clause set} ⇒ \text{term clause set} ⇒ \text{bool where}

resolution-deriv = rtranclp resolution-step

### 11 Soundness

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

evalsub \( E \ F \ σ = \text{eval}_{t} \ E \ F \circ \ σ \)

**lemma substitutiont**:
\[ \text{eval}_{t} \ E \ F \ (t \cdot t \ σ) = \text{eval}_{t} \ (\text{evalsub} \ E \ F \ σ) \ F \ t \]

**apply** (induction \( t \))

**unfolding** evalsub-def **apply** auto

**apply** (metis (mono-tags, lifting) comp-apply map-cong)

**done**

**lemma substitutionts**:
\[ \text{eval}_{ts} \ E \ F \ (ts \cdot ts \ σ) = \text{eval}_{ts} \ (\text{evalsub} \ E \ F \ σ) \ F \ ts \]

**using substitutiont by** auto

**lemma substitution**:
\[ \text{eval}_{l} \ E \ F \ G \ (l \cdot l \ σ) \leftrightarrow \text{eval}_{l} \ (\text{evalsub} \ E \ F \ σ) \ F \ G \ l \]

**apply** (induction \( l \))

**using substitutionts** **apply** (metis evall.simps(1) subl.simps(1))

**using substitutionts** **apply** (metis evall.simps(2) subl.simps(2))

**done**

**lemma subst-sound**:

**assumes asm**: \text{eval}_{c} \ F \ G \ C

**shows** \text{eval}_{c} \ F \ G \ (C \cdot \ls \ σ)

**unfolding** evalc-def **proof**
fix \( E \)
from asm have \( \forall E'. \exists l \in C. \text{eval}_l E' F G l \) using evalc-def by blast
then have \( \exists l \in C. \text{eval}_l (\text{evalsub} E F \sigma) F G l \) by auto
then show \( \exists l \in C \quad \gamma_s \quad \sigma. \text{eval}_l E F G l \) using substitution by blast
qed

lemma simple-resolution-sound:
assumes \( C_1 \text{sat} \): \( \text{eval}_c F G C_1 \)
assumes \( C_2 \text{sat} \): \( \text{eval}_c F G C_2 \)
assumes \( l_1 \text{inc}_1 \): \( l_1 \in C_1 \)
assumes \( l_2 \text{inc}_2 \): \( l_2 \in C_2 \)
assumes \( \text{comp} \): \( l_1^c = l_2 \)
shows \( \text{eval}_c F G ((C_1 - \{l_1\}) \cup (C_2 - \{l_2\})) \)
proof –
have \( \forall E. \exists l \in ((C_1 - \{l_1\}) \cup (C_2 - \{l_2\})). \text{eval}_l E F G l \)
proof
fix \( E \)
have \( \text{eval}_l E F G l_1 \lor \text{eval}_l E F G l_2 \) using comp by (cases \( l_1 \)) auto
then show \( \exists l \in ((C_1 - \{l_1\}) \cup (C_2 - \{l_2\})). \text{eval}_l E F G l \)
proof
assume \( \text{eval}_l E F G l_1 \)
then have \( \neg\text{eval}_l E F G l_2 \) using comp by (cases \( l_1 \)) auto
then have \( \exists l_2' \in C_2. l_2' \neq l_2 \land \text{eval}_l E F G l_2' \) using \( l_2 \text{inc}_2 \) \( C_2 \text{sat} \)
proof unfolding evalc-def by auto
then show \( \exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}). \text{eval}_l E F G l \) by auto
next
assume \( \text{eval}_l E F G l_2 \)
then have \( \neg\text{eval}_l E F G l_1 \) using comp by (cases \( l_1 \)) auto
then have \( \exists l_1' \in C_1. l_1' \neq l_1 \land \text{eval}_l E F G l_1' \) using \( l_1 \text{inc}_1 \) \( C_1 \text{sat} \)
proof unfolding evalc-def by auto
then show \( \exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}). \text{eval}_l E F G l \) by auto
qed
qed
then show ?thesis unfolding evalc-def by simp
qed

lemma mresolution-sound:
assumes \( sat_1 \): \( \text{eval}_c F G C_1 \)
assumes \( sat_2 \): \( \text{eval}_c F G C_2 \)
assumes \( \text{appl} \): applicable \( C_1 \quad C_2 \quad L_1 \quad L_2 \quad \sigma \)
shows \( \text{eval}_c F G (\text{mresolution} C_1 \quad C_2 \quad L_1 \quad L_2 \quad \sigma) \)
proof –
from \( sat_1 \) have \( sat_1 \sigma \): \( \text{eval}_c F G (C_1 \quad \gamma_s \quad \sigma) \) using subst-sound by blast
from \( sat_2 \) have \( sat_2 \sigma \): \( \text{eval}_c F G (C_2 \quad \gamma_s \quad \sigma) \) using subst-sound by blast
from \( \text{appl} \) obtain \( l_1 \) where \( l_1 \text{-p} \): \( l_1 \in L_1 \) unfolding applicable-def by auto
from \( l_1 \text{-p} \) \( \text{appl} \) have \( l_1 \in C_1 \) unfolding applicable-def by auto
then have \( \text{inc}_1 \sigma \): \( l_1 \quad \gamma \sigma \in C_1 \quad \gamma_s \quad \sigma \) by auto
from l1-p have unified1: l1 ∈ (L1 ∪ (L2^C)) by auto

from l1-p appl have l1 isl1σ: {l1 · σ} = L1 · l1σ
  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 · l2σ by auto

from l2-p have unified2: l2 ∈ (L1 ∪ (L2^C)) by auto

from unified1 unified2 appl have l1 · σ = (l2^C) · l1σ
  unfolding mgu1σ-def unifier1σ-def applicable-def by auto
then have comp: (l1 · σ)^C = l2 · l1σ 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 l2 isl2σ: {l2 · σ} = L2 · l2σ unfolding unifier1σ-def by auto

from sat1σ sat2σ inc1σ inc2σ comp have eval_c F G (\((C1 · l1σ) - \{l1 · σ\}\) ∪ \((C2 · l2σ) - \{l2 · l1σ\}\)) using simple-resolution-sound[of F G C1 · l1σ C2 · l2σ l1 · l1σ l2 · l1σ]
  by auto
from this l1 isl1σ l2 isl2σ 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: eval_c F G C
  shows eval_c F G C'
proof
  have ∀ E. ∃ l ∈ C'. eval_l E F G l
    proof
      fix E
      from sat have ∀ E. ∃ l ∈ C. eval_l E F G l unfolding eval_c-def by –
      then have ∃ l ∈ C. eval_l E F G l by auto
      then show ∃ l ∈ C'. eval_l E F G l using sup by auto
    qed
    then show eval_c F G C' unfolding eval_c-def by auto
  qed

43
theorem resolution-sound:
  assumes sat1: \( \text{eval}_c F G C_1 \)
  assumes sat2: \( \text{eval}_c F G C_2 \)
  assumes appl: applicable \( C_1 C_2 L_1 L_2 \sigma \)
  shows \( \text{eval}_c F G (\text{resolution} C_1 C_2 L_1 L_2 \sigma) \)
proof –
  from sat1 sat2 appl have \( \text{eval}_c F G (\text{mresolution} C_1 C_2 L_1 L_2 \sigma) \) using
  mresolution-sound by blast
  then show \( \text{thesis} \) using superset-sound resolution-superset by metis
qed

lemma mstep-sound:
  assumes mresolution-step Cs Cs'
  assumes eval cs F G Cs
  shows eval cs F G Cs'
using assms proof (induction rule: mresolution-step.induct)
case (mresolution-rule C_1 Cs C_2 l_1 l_2 \sigma)
  then have eval cs F G C_1 \land eval cs F G C_2 unfolding eval cs-def by auto
  then have eval cs F G (mresolution C_1 C_2 l_1 l_2 \sigma)
    using mresolution-sound mresolution-rule by auto
  then show \( \text{?case} \) using mresolution-rule unfolding eval cs-def by auto
next
case (standardize-apart C Cs C')
  then have eval cs F G unfolding eval cs-def by auto
  then have eval cs F G C' using subst-sound standardize-apart unfolding var-renaming-of-def
  instance-of1-def by metis
  then show \( \text{?case} \) using standardize-apart unfolding eval cs-def by auto
qed

theorem step-sound:
  assumes resolution-step Cs Cs'
  assumes eval cs F G Cs
  shows eval cs F G Cs'
using assms proof (induction rule: resolution-step.induct)
case (resolution-rule C_1 Cs C_2 l_1 l_2 \sigma)
  then have eval cs F G C_1 \land eval cs F G C_2 unfolding eval cs-def by auto
  then have eval cs F G (resolution C_1 C_2 l_1 l_2 \sigma)
    using resolution-sound resolution-rule by auto
  then show \( \text{?case} \) using resolution-rule unfolding eval cs-def by auto
next
case (standardize-apart C Cs C')
  then have eval cs F G unfolding eval cs-def by auto
  then have eval cs F G C' using subst-sound standardize-apart unfolding var-renaming-of-def
  instance-of1-def by metis
  then show \( \text{?case} \) using standardize-apart unfolding eval cs-def by auto
qed

lemma mderivation-sound:
  assumes mresolution-deriv Cs Cs'
  assumes eval cs F G Cs
  shows eval cs F G Cs'
using assms proof (induction rule: resolution-step.induct)
case (resolution-rule C_1 Cs C_2 l_1 l_2 \sigma)
  then have eval cs F G C_1 \land eval cs F G C_2 unfolding eval cs-def by auto
  then have eval cs F G (resolution C_1 C_2 l_1 l_2 \sigma)
    using resolution-sound resolution-rule by auto
  then show \( \text{?case} \) using resolution-rule unfolding eval cs-def by auto
next
case (standardize-apart C Cs C')
  then have eval cs F G unfolding eval cs-def by auto
  then have eval cs F G C' using subst-sound standardize-apart unfolding var-renaming-of-def
  instance-of1-def by metis
  then show \( \text{?case} \) using standardize-apart unfolding eval cs-def by auto
qed
assumes \( eval_{cs} F G Cs \)
shows \( eval_{cs} F G Cs' \)
using assms unfolding mresolution-deriv-def
proof (induction rule: rtranclp.induct)
  case rtrancl-refl then show ?case by auto
next
  case (rtrancl-into-rtrancl Cs_1 Cs_2 Cs_3) then show ?case using mstep-sound by auto
qed

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

theorem derivation-sound-refute:
  assumes deriv: resolution-deriv Cs Cs' \& \{\} \in Cs'
  shows \( \neg eval_{cs} F G Cs \)
proof
  from deriv have \( eval_{cs} F G Cs \rightarrow eval_{cs} F G Cs' \) using derivation-sound by auto
  moreover
  from deriv have \( eval_{cs} F G Cs' \rightarrow eval_{c} F G \{\} \) unfolding eval_{cs}-def by auto
  moreover
  then have \( eval_{c} F G \{\} \rightarrow False \) unfolding eval_{c}-def by auto
  ultimately show ?thesis by auto
qed

12 Herbrand Interpretations

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

term \( HFun \)

lemma eval-ground:\`
  assumes ground_t t
  shows \( (eval_t E HFun t) = hterm-of-fterm t \)
using assms by (induction t) auto

45
lemma eval-ground\_ts:
assumes ground\_ts \, ts
shows (eval\_ts E HFun ts) = hterms-of-fterms ts
unfolding hterms-of-fterms-def using assms eval-ground\_ts by (induction ts) auto

lemma eval\_l-ground\_ts:
assumes asm: ground\_ts 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 E HFun ts) by auto
also have \ldots = 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

\begin{align*}
& \text{ground\_l l} \\
& \land (\text{let } i = \text{nat-of-fatom (get-atom l) in} \\
& \quad i < \text{length G} \land G ! i = (\neg \text{sign l})
\end{align*}

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

abbreviation falsifies\_g :: partial-pred-denot \Rightarrow fterm clause \Rightarrow bool where
falsifies\_g G C \equiv ground\_ts C \land (\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 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 (\text{let } n = \text{nat-of-hatom (P, ts) in} \\
\quad f (\text{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 \mapsto (sub-of-denot E))
by (induction t) (auto simp add: ground-fterm-of-hterm)
lemma ground-sub-of-denotts: ground\textsubscript{ts} (ts \cdot \text{sub-of-denot} E)
using ground-sub-of-denott by simp

lemma ground-sub-of-denottl: ground\textsubscript{l} (l \cdot \text{sub-of-denot} E)
proof
  have ground\textsubscript{ts} (subs (get-terms l) (sub-of-denot E))
  using ground-sub-of-denotts by auto
  then show ?thesis by (cases l) auto
qed

lemma sub-of-denot-equivx: eval\textsubscript{t} E HFun (sub-of-denot E x) = E x
proof
  have ground\textsubscript{ts} (sub-of-denot E x)
  using ground-fterm-of-hterm by simp
  then have eval\textsubscript{t} E HFun (sub-of-denot E x) = hterm-of-fterm (sub-of-denot E x)
  using eval-ground\textsubscript{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\textsubscript{t} E HFun (t \cdot \text{sub-of-denot} E) = eval\textsubscript{ts} E HFun t
using sub-of-denot-equivx by (induction t) auto

lemma sub-of-denot-equivts: eval\textsubscript{ts} E HFun (ts \cdot \text{sub-of-denot} E) = eval\textsubscript{ts} E HFun ts
using sub-of-denot-equivt by simp

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

47
show ?case by blast
qed

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

lemma sub-of-denot-eq-equiv-ground:
  \( \text{eval}_{l \cdot l} E \ HFun G \ l = \text{eval}_{l \cdot l} E \ HFun G \ (l \cdot l \text{ sub-of-denot } E) \land \text{ground}_{l \cdot l} (l \cdot l \text{ sub-of-denot } E) \)
  using sub-of-denot-eq-equiv-ground by auto

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

lemma partial-equiv-subst:
  assumes falsifies \(_C\) \(_G\) \(_C_1\)
  shows falsifies \(_C\) \(_G\) \(_C_1\)
proof
  from assms obtain \(_C'\) where \(_C'\cdot\_p\): instance-of_{\_s} \(_C'\) (\(_C'\cdot\_\_s\) \(_\tau\)) \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} \(_C'\cdot\_C\) using \(_C'\cdot\_p\) instance-of_{\_s-trans} by auto
  then show \(?thesis\) using \(_C'\cdot\_p\) by auto
qed

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

lemma sub-of-denot-eq-equiv-ground:
  \((\exists l \in C. \text{eval}_{l \cdot l} E \ HFun G \ l) \leftrightarrow (\exists l \in C \cdot l \text{ sub-of-denot } E. \text{eval}_{l \cdot l} E \ HFun G \ l))
  \land \text{ground}_{l \cdot l} (C \cdot l \text{ sub-of-denot } E)
  using sub-of-denot-eq-equiv-ground' by auto

lemma std1-falsifies: falsifies_{\_s} \(_G\) \(_C_1\) \(<\leftrightarrow\> falsifies_{\_s} \(_G\) (std1 \(_C_1\))
proof
  assume asm: falsifies_{\_s} \(_G\) \(_C_1\)
  then obtain \(_C_2\) where instance-of_{\_s} \(_C_2\) \(_C_1\) \land falsifies_{\_g} \(_G\) \(_C_2\) by auto
  moreover
  then have instance-of_{\_s} \(_C_2\) (std1 \(_C_1\)) using std-apart-instance-of_{\_s}1 instance-of_{\_s-trans} by blast
  ultimately
  show falsifies_{\_s} \(_G\) (std1 \(_C_1\)) by auto
next
  assume asm: falsifies_{\_s} \(_G\) (std1 \(_C_1\))
  then have inst: instance-of_{\_s} (std1 \(_C_1\)) \(_C_1\) unfolding instance-of_{\_s-def} by auto

  from asm obtain \(_C_2\) where instance-of_{\_s} \(_C_2\) (std1 \(_C_1\)) \land falsifies_{\_g} \(_G\) \(_C_2\) by auto
  moreover

48
then have \( \text{instance-of}_{1s} C_g C_1 \) using \( \text{inst} \) \( \text{instance-of}_{1s} \text{-trans} \) by blast
ultimately
show \( \text{falsifies}_c G C_1 \) by auto
qed

lemma \( \text{std}_2\text{-falsifies}: \text{falsifies}_c G C_2 \iff \text{falsifies}_c G (\text{std}_2 C_2) \)
proof
assume asm: \( \text{falsifies}_c G C_2 \)
then obtain \( C_g \) where \( \text{instance-of}_{1s} C_g C_2 \land \text{falsifies}_g G C_g \) by auto
moreover
then have \( \text{instance-of}_{1s} C_g (\text{std}_2 C_2) \) using \( \text{std}\text{-apart-instance-of}_{1s} \text{2 instance-of}_{1s} \text{-trans} \) by blast
ultimately
show \( \text{falsifies}_c G (\text{std}_2 C_2) \) by auto
next
assume asm: \( \text{falsifies}_c G (\text{std}_2 C_2) \)
then have \( \text{inst} \): \( \text{instance-of}_{1s} (\text{std}_2 C_2) C_2 \) unfolding \( \text{instance-of}_{1s} \text{-def} \) by auto
from asm obtain \( C_g \) where \( \text{instance-of}_{1s} C_g (\text{std}_2 C_2) \land \text{falsifies}_g G C_g \) by auto
moreover
then have \( \text{instance-of}_{1s} C_g C_2 \) using \( \text{inst} \) \( \text{instance-of}_{1s} \text{-trans} \) by blast
ultimately
show \( \text{falsifies}_c G C_2 \) by auto
qed

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

lemma \( \text{std}_2\text{-renames}: \text{var-renaming-of} C_2 (\text{std}_2 C_2) \)
proof
−
have \( \text{instance-of}_{1s} C_2 (\text{std}_2 C_2) \) using \( \text{std}\text{-apart-instance-of}_{1s} \text{2} \) by auto
moreover have \( \text{instance-of}_{1s} (\text{std}_2 C_2) C_2 \) unfolding \( \text{instance-of}_{1s} \text{-def} \) by auto
ultimately show \( \text{var-renaming-of} C_2 (\text{std}_2 C_2) \) unfolding \( \text{var-renaming-of-def} \) by auto
qed

14 Semantic Trees

abbreviation closed-branch :: partial-pred-denot \( \Rightarrow \) tree \( \Rightarrow \) fterm clause set \( \Rightarrow \) bool where
closed-branch \( G T Cs \equiv \text{branch} \ G T \land \text{falsifies}_{cs} \ G Cs \)

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

open-branch \( G T Cs \equiv \text{branch} \ G T \land \neg \text{falsifies}_{cs} \ G Cs \)

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

closed-tree \( T Cs \iff \text{anybranch} \ T (\lambda b. \text{closed-branch} \ b T Cs) \land \text{anyinternal} \ T (\lambda p. \neg \text{falsifies}_{cs} \ p Cs) \)

15 Herbrand’s Theorem

**lemma** maximum:

assumes \( \text{asm} : \text{finite} \ C \)

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

**proof**

from \( \text{asm} \) show \( \forall l \in C. \ f l \leq (\text{Max} \ (f \ C)) \) by auto

qed

**lemma** extend-preserves-model:

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

assumes \( \text{C-ground} : \text{ground}_{ls} C \)

assumes \( \text{C-sat} : \neg \text{falsifies}_{c} (f (\text{Suc} \ n)) C \)

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

shows \( \text{eval}_c \ HFun (\text{extend} \ f) C \)

**proof**

let \(?F = HFun\

let \(?G = \text{extend} f\

\{ \)

fix \( E \)

from \( \text{C-sat} \) have \( \forall C'. (\neg \text{instance-of}_{ls} C' C \lor \neg \text{falsifies}_g (f (\text{Suc} \ n)) C') \) by auto

then have \( \neg \text{falsifies}_g (f (\text{Suc} \ n)) C \) using \( \text{instance-of}_{ls}\text{-self} \) by auto

then obtain \( i \) where \( l-p. \ l \in C \land \neg \text{falsifies}_l (f (\text{Suc} \ n)) l \) using \( \text{C-ground} \) by blast

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

from \( l-p \) have \( i-n : ?i \leq n \) using \( n\text{-max} \) by auto

then have \( j-n : ?i < \text{length} (f (\text{Suc} \ n)) \) using \( f\text{-infpath} \text{infpath-length[of f]} \) by auto

have \( \text{eval}_l E \ HFun (\text{extend} f) l \)

**proof** (cases \( l \))

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

from \( \text{Pos} \ l-p \) \( \text{C-ground} \) have \( \text{ts-ground} : \text{ground}_{ls} \) \( \text{ts} \) by auto

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

then have \( f (\text{Suc} \ n) \! ?i = \text{True} \) using \( j-n \) \( \text{Pos \ ts-ground \ empty-subts[of ts]} \) unfolding \( \text{falsifies}_{l}\text{-def} \) by auto

50
moreover have \( f \left( \text{Suc } ?i \right) ! ?i = f \left( \text{Suc } n \right) ! ?i \)
using \( f \)-infpath \( i-n \) infpath-length[of \( f \)] ith-in-extension[of \( f \)] by simp
ultimately
have \( f \left( \text{Suc } ?i \right) ! ?i = \text{True} \) using Pos by auto
then have \( ?G P \left( \text{hterms-of-fterms ts} \right) \) using Pos by (simp add: nat-of-fatom-def)

then show \( ?\text{thesis using } eval_{1-ground_{ts}}[\text{of ts - } \text{?G P}] \) ts-ground Pos by auto
next
case \( \text{(Neg P ts)} \)
from \( \text{Neg l-p C-ground} \) have ts-ground: ground ts ts by auto
have \( \neg \text{falsifies } l \left( f \left( \text{Suc } n \right) \right) \) using l-p by auto
then have \( f \left( \text{Suc } \text{n} \right) ! ?i = \text{False} \) using j-n Neg ts-ground empty-subts[of ts] unfolding falsifies-def by auto
moreover have \( f \left( \text{Suc } ?i \right) ! ?i = f \left( \text{Suc } n \right) ! ?i \)
using \( f \)-infpath \( i-n \) j-n infpath-length[of \( f \)] ith-in-extension[of \( f \)] by simp
ultimately
have \( f \left( \text{Suc } ?i \right) ! ?i = \text{False} \) using Neg by auto
then have \( \neg ?G P \left( \text{hterms-of-fterms ts} \right) \) using Neg by (simp add: nat-of-fatom-def)

then show \( ?\text{thesis using } Neg \text{ eval}_{1-ground_{ts}}[\text{of ts - } \text{?G P}] \) ts-ground by auto
qed

proof
then have \( \exists l \in C. \text{eval}_{l} E HFun \left( \text{extend } f \right) \) l using l-p by auto
\}
then have eval_{c} HFun \left( \text{extend } f \right) C unfolding eval_{c}-def by auto
then show \( ?\text{thesis using } instance-of_{ts-self} \) by auto
qed

lemma \text{extend-preserves-model2}:
assumes \( \text{f-infpath: wf-infpath } \left( f :: \text{nat } \Rightarrow \text{partial-pred-denot} \right) \)
assumes \( \text{C-ground: ground}_{1s} C \)
assumes \( \text{fin-c: finite } C \)
assumes \( \text{model-C: } \forall n. \neg \text{falsifies}_{c} \left( f \left( n \right) \right) C \)
shows \( \text{C-false: eval}_{c} HFun \left( \text{extend } f \right) C \)

proof
— Since \( C \) is finite, \( C \) has a largest index of a literal.
obtain \( n \) where largest: \( \forall l \in C. \text{nat-of-fatom } \left( \text{get-atom } l \right) \leq n \) using fin-c
maximum[of \( C \) \( \lambda l. \text{nat-of-fatom } \left( \text{get-atom } l \right) \)] by blast
moreover
then have \( \neg \text{falsifies}_{c} \left( f \left( \text{Suc } n \right) \right) C \) using model-C by auto
ultimately show \( ?\text{thesis using } \text{model-C f-infpath C-ground extend-preserves-model}[\text{of } f \left( C \left( n \right) \right)] \) by blast
qed

lemma \text{extend-infpath}:
assumes \( \text{f-infpath: wf-infpath } \left( f :: \text{nat } \Rightarrow \text{partial-pred-denot} \right) \)
assumes \( \text{model-c: } \forall n. \neg \text{falsifies}_{c} \left( f \left( n \right) \right) C \)
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 ·_σ sub-of-denot E) by auto
  have groundcσ: groundcσ (C ·_σ sub-of-denot E) using sub-of-denot-equiv-ground
  by auto

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

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

lemma inpath-model:
  assumes f-infpath: wf-infpath (f :: nat ⇒ partial-pred-denot)
  assumes model-cs: ∀ n. ¬ falsifies_c (f n) Cs
  assumes fin-cs: finite Cs
  assumes fin-c: ∀ C ∈ Cs. finite C
  shows eval_c HFun (extend f) Cs
proof –
  let ?F = HFun

  have ∀ C ∈ Cs. eval_c ?F (extend f) C
    proof (rule ballI)
      fix C
      assume asm: C ∈ Cs
      then have ∀ n. ¬ falsifies_c (f n) C using model-cs by auto
      then show eval_c ?F (extend f) C using fin-c asm f-infpath extend-infpath[of f C] by auto
      qed
    then show eval_c ?F (extend f) Cs unfolding eval_c-def by auto
    qed

fun deeftree :: nat ⇒ tree where
deeftree 0 = Leaf
lemma \textit{branch-length}:
\begin{itemize}
\item assumes \textit{branch b (deeptree n)}
\item shows \textit{length b = n}
\item using \textit{assms proof (induction n arbitrary: b)}
\item case 0 then show \textit{?case using branch-inv-Leaf by auto}
\item next
\item case (Suc n) then have \textit{branch b (Branching (deeptree n) (deeptree n)) by auto}
\item then obtain a b' where \textit{p: b = a#b' \& branch b' (deeptree n) using branch-inv-Branching[of b] by blast}
\item then have \textit{length b' = n using Suc by auto}
\item then show \textit{?case using p by auto}
\end{itemize}
\textbf{qed}

lemma \textit{infinity}:
\begin{itemize}
\item assumes \textit{inj: \forall n :: nat. undiago (diago n) = n}
\item assumes \textit{all-tree: \forall n :: nat. (diago n) \in tree}
\item shows \textit{\neg finite tree}
\item proof –
\item from \textit{inj all-tree have \forall n. n = undiago (diago n) \& (diago n) \in tree by auto}
\item then have \textit{\forall n, \exists ds. n = undiago ds \& ds \in tree by auto}
\item then have \textit{undiago \cdot tree = (UNIV :: nat set) by auto}
\item then have \textit{\neg finite tree by (metis finite-imageI infinite-UNIV-nat)}
\item then show \textit{?thesis by auto}
\end{itemize}
\textbf{qed}

lemma \textit{longer-falsifies_{1}}:
\begin{itemize}
\item assumes \textit{falsifies \_ds l}
\item shows \textit{falsifies_{1} (ds@d) l}
\item proof –
\item let \textit{?i = nat-of-fatom (get-atom l)}
\item from \textit{assms have i-p: groundl l \& \ ?i < length ds \& ds ! ?i = (\neg\textit{sign l}) unfolding falsifies_{1}-def by meson}
\item moreover
\item from \textit{i-p have \ ?i < length (ds@d) by auto}
\item moreover
\item from \textit{i-p have (ds@d) ! \ ?i = (\neg\textit{sign l}) by (simp add: nth-append)}
\item ultimately
\item show \textit{?thesis unfolding falsifies_{1}-def by simp}
\end{itemize}
\textbf{qed}

lemma \textit{longer-falsifies_{9}}:
\begin{itemize}
\item assumes \textit{falsifies_{9} ds C}
\item shows \textit{falsifies_{9} (ds @ d) C}
\item proof –
\item \begin{itemize}
\item fix l
\end{itemize}
\end{itemize}
\textbf{qed}
assume \( l \in C \)
then have falsifies\(_l\) (\( ds \at d \)) \( l \) using assms longer-falsifies\(_l\) by auto
\)
then show \( \text{thesis} \) using assms by auto
qed

theorem herbrand':
assumes openb: \( \forall T. \exists G. \text{open-branch } G T Cs \)
assumes finite-cs: finite Cs \( \forall C \in Cs. \text{finite } C \)
shows \( \exists G. \text{eval}_c s HFun G Cs \)
proof –
— Show \( T \) infinite:
let \( \text{?tree} = \{ G. \neg \text{falsifies}_c s G Cs \} \)
let \( \text{?undiag} = \text{length} \)
let \( \text{?diag} = (\lambda l. \text{SOME } b. \text{open-branch } b (\text{deeptree } l) Cs) :: \text{nat} \Rightarrow \text{partial-pred-denot} \)
from openb have diag-open: \( \forall l. \text{open-branch } (\text{?diag } l) (\text{deeptree } l) Cs \)
using someI-ex[of \( \lambda b. \text{open-branch } b (\text{deeptree } -) Cs \)] by auto
then have \( \forall n. \text{?undiag} (\text{?diag } n) = n \) using branch-length by auto
moreover
have \( \forall n. (\text{?diag } n) \in \text{?tree} \) using diag-open by auto
ultimately
have \( \neg \text{finite } \text{?tree} \) using infinity[of - \( \lambda n. \text{SOME } b. \text{open-branch } b (- n) Cs \)] by simp
— Get infinite path:
moreover
have \( \forall ds \text{ } d. \neg \text{falsifies}_{cs} (ds \circ d) \text{ } Cs \rightarrow \neg \text{falsifies}_{cs} \text{ } ds \text{ } Cs \)

using longer-falsifies[of Cs] by blast

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

ultimately

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

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

— Apply above infpath lemma:

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

qed

lemma shorter-falsifies_l:

assumes falsifies_l (ds@d) l

assumes nat-of-fatom (get-atom l) < length ds

shows falsifies_l ds l

proof –

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

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

unfolding falsifies_l-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 \(?\text{thesis}\) using assms unfolding falsifies_l-def by simp

qed

theorem herbrand′-contra:

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

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

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

proof –

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

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

qed

theorem herbrand:

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

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

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

proof –

from unsat finite-cs obtain T where anybranch T (\(\lambda b. \text{closed-branch } b \text{ } T \text{ } Cs\))

using herbrand′-contra[of Cs] by blast

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

using cutoff-branch-internal[of T \(\lambda p. \text{falsifies}_{cs} \text{ } p \text{ } Cs\)] by blast

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

qed

end
16  Lifting Lemma

theory Completeness imports Resolution begin

locale unification =
  assumes unification: \( \forall \sigma. L. \text{finite } L \Longrightarrow \text{unifier}_{ts} \sigma L \Longrightarrow \exists \vartheta. \text{mgu}_{ts} \vartheta L \)
begin

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

lemma lifting:
  assumes fin: \( \text{finite } C_1 \land \text{finite } C_2 \)
  assumes apart: \( \text{vars}_{ts} C_1 \cap \text{vars}_{ts} C_2 = \{\} \)
  assumes inst: \( \text{instance-of}_{ts} C_1' C_1 \land \text{instance-of}_{ts} C_2' C_2 \)
  assumes appl: \( \text{applicable } C_1' C_2' L_1' L_2' \sigma \)
  shows \( \exists L_1 L_2 \tau. \text{applicable } C_1 C_2 L_1 L_2 \tau \land
given \text{instance-of}_{ts} (\text{resolution } C_1' C_2' L_1' L_2' \sigma) \) \( (\text{resolution } C_1 C_2 L_1 L_2 \tau) \)
proof -
  — Obtaining the subsets we resolve upon:
  let \( ?R_1' = C_1' - L_1' \) and \( ?R_2' = C_2' - L_2' \)

  from \( \text{inst} \) obtain \( \gamma \mu \) where \( C_1' \gamma_{ts} \gamma = C_1' \land C_2' \gamma_{ts} \mu = C_2' \)
  unfolding \( \text{instance-of}_{ts}\text{-def} \) by auto
  then obtain \( \eta \) where \( \eta\text{-p: } C_1' \gamma_{ts} \eta = C_1' \land C_2' \gamma_{ts} \eta = C_2' \)
  using \( \text{apart merge-sub by force} \)

  from \( \eta\text{-p} \) obtain \( L_1 \) where \( L_1\text{-p: } L_1 \subseteq C_1 \land L_1' \gamma_{ts} \eta = L_1' \land (C_1 - L_1) \gamma_{ts} \eta = ?R_1' \)
  using \( \text{appl project-sub using applicable-def by metis} \)
let \( ?R_1 = C_1 - L_1 \)

  from \( \eta\text{-p} \) obtain \( L_2 \) where \( L_2\text{-p: } L_2 \subseteq C_2 \land L_2' \gamma_{ts} \eta = L_2' \land (C_2 - L_2) \gamma_{ts} \eta = ?R_2' \)
  using \( \text{appl project-sub using applicable-def by metis} \)
let \( ?R_2 = C_2 - L_2 \)

  — Obtaining substitutions:
  from \( \text{appl have mgu}_{ts} \sigma (L_1' \cup L_2 \subseteq C_2' \gamma_{ts} \eta) \) using \( \text{applicable-def by auto} \)
  then have \( \text{mgu}_{ts} \sigma ((L_1\gamma_{ts} \eta) \cup (L_2' \gamma_{ts} \eta)) \) using \( \text{L_1-p L_2-p by auto} \)
  then have \( \text{mgu}_{ts} \sigma ((L_1 \cup L_2' \subseteq C_2' \gamma_{ts} \eta)) \) using \( \text{compls-subs subs-anion by auto} \)
  then have \( \text{unifier}_{ts} \sigma ((L_1 \cup L_2' \subseteq C_2' \gamma_{ts} \eta)) \) using \( \text{mgu}_{ts}\text{-def by auto} \)
  then have \( \eta\text{un: } \text{unifier}_{ts} (\eta \cdot \sigma) (L_1 \cup L_2' \subseteq C_2' \gamma_{ts} \eta) \)
  using \( \text{unifier}_{ts}\text{-def composition-conseq2l by auto} \)

  then obtain \( \tau \) where \( \tau\text{-p: } \text{mgu}_{ts} \tau (L_1 \cup L_2' \subseteq C_2' \gamma_{ts} \eta) \)
  using \( \text{unification fin L_1-p L_2-p by meson finite-UnI finite-image rev-finite-subset} \)
  then obtain \( \varphi \) where \( \varphi\text{-p: } \tau \cdot \varphi = \eta \cdot \sigma \) using \( \eta\text{un mgu}_{ts}\text{-def by auto} \)

  — Showing that we have the desired resolvent:
  let \( \exists C = ((C_1 - L_1) \cup (C_2 - L_2)) \gamma_{ts} \tau \)
have \(?C \cdot _{\text{ls}} \varphi = (\mathcal{R}_1 \cup \mathcal{R}_2) \cdot _{\text{ls}} (\tau \cdot \varphi)\)

using subls-union composition-conseq2ls by auto

also have ... = (\mathcal{R}_1 \cup \mathcal{R}_2) \cdot _{\text{ls}} (\eta \cdot \sigma) using \varphi - p by auto

also have ... = ((\mathcal{R}_1 \cdot _{\text{ls}} \eta) \cup (\mathcal{R}_2 \cdot _{\text{ls}} \eta)) \cdot _{\text{ls}} \sigma

using subls-union composition-conseq2ls by auto

also have ... = (\mathcal{R}_1 \cup \mathcal{R}_2) \cdot _{\text{ls}} (\eta \cdot \sigma)

using \varphi - p \text{ by auto}

finally have \(?C \cdot _{\text{ls}} \varphi = ((\mathcal{C}_1' \cdot _{\text{ls}} \mathcal{L}_1) \cup (\mathcal{C}_2' \cdot _{\text{ls}} \mathcal{L}_2)) \cdot _{\text{ls}} \sigma\) by auto

using res: instance-of1s (resolution \(\mathcal{C}_1' \cdot _{\text{ls}} \mathcal{L}_1 \cdot _{\text{ls}} \mathcal{L}_2') \cdot _{\text{ls}} \sigma\) by auto

— Showing that the resolution rule is applicable:

have \(\mathcal{C}_1' \neq \emptyset \land \mathcal{C}_2' \neq \emptyset \land \mathcal{L}_1' \neq \emptyset \land \mathcal{L}_2' \neq \emptyset\)

using appl applicable-def by auto

then have \(\mathcal{C}_1 \neq \emptyset \land \mathcal{C}_2 \neq \emptyset \land \mathcal{L}_1 \neq \emptyset \land \mathcal{L}_2 \neq \emptyset\)

using \eta - p \(\mathcal{L}_1\) - p \(\mathcal{L}_2\) - p by auto

then have appl: applicable \(\mathcal{C}_1 \cdot _{\text{ls}} \mathcal{C}_2 \cdot _{\text{ls}} \mathcal{L}_1 \cdot _{\text{ls}} \mathcal{L}_2 \cdot _{\text{ls}} \tau\)

using apart \(\mathcal{L}_1\) - p \(\mathcal{L}_2\) - p \(\tau\) - p applicable-def by auto

from ins appl show \(?\text{thesis}\) by auto

qed

17 Completeness

lemma falsifies$_g$-empty:

assumes falsifies$_g \mathcal{\[} C$

shows \(C = \emptyset\)

proof —

have \(\forall l \in C. \text{False}\)

proof

fix l

assume \(l \in C\)

then have falsifies$_1 \mathcal{\[} l using assms by auto

then show \text{False unfolding falsifies$_1$-def by (cases l) auto}

qed

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

qed

lemma falsifies$_c$-empty:

assumes falsifies$_c \mathcal{\[} C$

shows \(C = \emptyset\)

proof —

from assms obtain \(C'\) where \(C'\) - p: instance-of$_{1s}$ \(C'\) - p \(C \land \text{falsifies$_g \mathcal{\[} C'\}$

by auto

then have \(C' = \emptyset\) using falsifies$_g$-empty by auto

then show \(C = \emptyset\) using \(C'\) - p unfolding instance-of$_{1s}$-def by auto

qed

lemma complements-do-not-falsify':

57
assumes $l_1 \in C_1'$
assumes $l_2 \in C_1'$
assumes $l_1 = l_2$
assumes fals: falsifies $G C_1'$
shows False
proof (cases $l_1$)
  case (Pos $p$ $ts$)
  let $?i1 = \text{nat-of-fatom} (p, ts)$

    from assms have gr: ground $l_1$ unfolding falsifies$_1$-def by auto
    then have Neg: $l_2 = \text{Neg p ts}$ using comp Pos by (cases $l_2$) auto
    from falsif have falsifies $G l_1$ using $l1C1'$ by auto
    then have $G \neg {?i1} = \text{False}$ using $l1C1'$ Pos unfolding falsifies$_1$-def by (induction $Pos p$ $ts$) auto
    moreover
    let $?i2 = \text{nat-of-fatom} (\text{get-atom} l_2)$
    from falsif have falsifies$_1$ $G l_2$ using $l_2C1'$ by auto
    then have $G \neg {?i2} = (\neg \text{sign} l_2)$ unfolding falsifies$_1$-def by meson
    then have $G \neg {?i1} = (\neg \text{sign} l_2)$ using Pos Neg comp by simp
    then have $G \neg {?i1} = \text{True}$ using Neg by auto
    ultimately show $?\text{thesis by auto}$
  next
  case (Neg $p$ $ts$)
  let $?i1 = \text{nat-of-fatom} (p, ts)$

    from assms have gr: ground $l_1$ unfolding falsifies$_1$-def by auto
    then have Pos: $l_2 = \text{Pos p ts}$ using comp Neg by (cases $l_2$) auto
    from falsif have falsifies$_1$ $G l_1$ using $l1C1'$ by auto
    then have $G \neg {?i1} = \text{True}$ using $l1C1'$ Neg unfolding falsifies$_1$-def by (metis get-atom.simps(2) literal.disc(2))
    moreover
    let $?i2 = \text{nat-of-fatom} (\text{get-atom} l_2)$
    from falsif have falsifies$_1$ $G l_2$ using $l_2C1'$ by auto
    then have $G \neg {?i2} = (\neg \text{sign} l_2)$ unfolding falsifies$_1$-def by meson
    then have $G \neg {?i1} = (\neg \text{sign} l_2)$ using Pos Neg comp by simp
    then have $G \neg {?i1} = \text{False}$ using Pos using literal.disc(1) by blast
    ultimately show $?\text{thesis by auto}$

qed

lemma complements-do-not-falsify:
  assumes $l1C1': l_1 \in C_1'$
  assumes $l2C1': l_2 \in C_1'$
  assumes fals: falsifies $G C_1'$
  shows $l_1 \neq l_2$
  using assms complements-do-not-falsify' by blast

lemma other-falsified:
assumes $C_1'$-p: $\text{ground}_{1_\text{s}} C_1' \land \text{falsifies}_{g} (B @ [d]) C_1'$
assumes $l$-p: $l \in C_1'$ $\land \text{nat-of-fatom} (\text{get-atom} l) = \text{length} B$
assumes other: $lo \in C_1'$ $lo \neq l$
shows $\text{falsifies}_1 B lo$

proof

let $?i = \text{nat-of-fatom} (\text{get-atom} lo)$

have $\text{ground}_{l_2}$: $\text{ground}_l$ $l$ using $l$-p $C_1'$-p by auto
  — They are, of course, also ground:
  
  have $\text{ground-lo}$: $\text{ground}_l$ $lo$ using $l$-p by auto

  assumes $l$-p $C_1'$-p other by auto

  from $C_1'$-p have $\text{falsifies}_g (B @ [d]) (C_1' - \{l\})$ by auto
  — And indeed, falsified by $B @ [d]$:
  then have $\text{loB}_2$: $\text{falsifies}_1 (B @ [d]) lo$ using other by auto

  then have $?i < \text{length} (B @ [d])$ unfolding $\text{falsifies}_{\text{def}}$ by meson
  — And they have numbers in the range of $B @ [d]$, i.e. less than $\text{length} B + 1$:
  then have $\text{nat-of-fatom} (\text{get-atom} lo) < \text{length} B + 1$ using $\text{undia-diag-fatom}$

  — And indeed, falsified by $B @ [d]$:
  then have $\text{loB}_2$: $\text{falsifies}_1 (B @ [d]) lo$ using other by auto

  then have $?i < \text{length} (B @ [d])$ unfolding $\text{falsifies}_{\text{def}}$ by meson

  — And they have numbers in the range of $B @ [d]$, i.e. less than $\text{length} B + 1$:
  then have $\text{nat-of-fatom} (\text{get-atom} lo) < \text{length} B + 1$ using $\text{undia-diag-fatom}$

moreover

have $l$-lo: $l \neq lo$ using other by auto
  — The are not the complement of $l$, since then the clause could not be falsified:
  
  have $lc$-lo: $lo \neq l$ using $l$-p other complements-do-not-falsify[of $lo$ $C_1'$ $l$

  from $l$-lo $lc$-lo have $\text{get-atom} l \neq \text{get-atom} lo$ using $\text{sign-comp-atom}$ by metis

  then have $\text{nat-of-fatom} (\text{get-atom} lo) \neq \text{nat-of-fatom} (\text{get-atom} l)$
  using $\text{bij-betw-def}$ $\text{inj-on-def}$ by metis

  — Therefore they have different numbers:
  then have $\text{nat-of-fatom} (\text{get-atom} lo) \neq \text{length} B$ using $l$-p by auto

  ultimately
  — So their numbers are in the range of $B$:
  
  have $\text{nat-of-fatom} (\text{get-atom} lo) < \text{length} B$ by auto

  — So we did not need the last index of $B @ [d]$ to falsify them, i.e. $B$ suffices:
  then show $\text{falsifies}_1 B lo$ using $\text{loB}_2$ shorter-falsifies$_1$ by blast

qed

theorem completeness,'

assumes closed-tree $T$ Cs
assumes $\forall C \in Cs$. finite $C$
shows $\exists Cs'$. resolution-deriv $Cs$ $Cs' \land \{\} \in Cs'$

using assms proof (induction $T$ arbitrary: $Cs$ rule: measure-induct-rule[of tree-size])

fix $T :: tree$
fix $Cs :: \text{fterm clause set}$
assume ih: $\forall T' Cs$. tressize $T' < \text{tressize} T \Rightarrow$ closed-tree $T'$ $Cs \Rightarrow$

$\forall C \in Cs$. finite $C \Rightarrow \exists Cs'$. resolution-deriv $Cs$ $Cs' \land \{\} \in Cs'$

assume clo: closed-tree $T$ Cs
assume finite-Cs: $\forall C \in Cs$. finite $C$
  { — Base case:
    assume tressize $T = 0$}
then have $T = \text{Leaf}$ using treesize-Leaf by auto
then have closed-branch $\emptyset$ Leaf Cs using branch-inv-Leaf clo unfolding closed-tree-def by auto
then have falsifies, $\emptyset$ Cs by auto
then have $\{\} \in Cs$ using falsifies,empty by auto
then have $\exists Cs', \text{resolution-deriv} Cs Cs' \land \{\} \in Cs'$ unfolding resolution-deriv-def by auto

moreover

$\{\}$ — Induction case:
assume treesize $T > 0$
then have $\exists l \ r. T = \text{Branching} l \ r$ (cases $T$) auto

— Finding sibling branches and their corresponding clauses:
then obtain $B$ where $b-p$:

using internal-branch[of - $\emptyset$ - $T$] Branching-Leaf-Leaf-Tree by fastforce
let $?B_1 = B@[\text{True}]$
let $?B_2 = B@[\text{False}]$

obtain $C_{1,o}$ where $C_{1,o} \in Cs \land \text{falsifies}_c ?B_1 C_{1,o}$ using $b-p$ clo unfolding closed-tree-def by metis
obtain $C_{2,o}$ where $C_{2,o} \in Cs \land \text{falsifies}_c ?B_2 C_{2,o}$ using $b-p$ clo unfolding closed-tree-def by metis

— Standardizing the clauses apart:
let $?C_1 = \text{std}_1 C_{1,o}$
let $?C_2 = \text{std}_2 C_{2,o}$
have $C_{1,p}$: falsifies, $?B_1 ?C_1$ using $\text{std}_1$-falsifies $C_{1,o}$-p by auto
have $C_{2,p}$: falsifies, $?B_2 ?C_2$ using $\text{std}_2$-falsifies $C_{2,o}$-p by auto

have fin: finite $?C_1 \land \text{finite } ?C_2$ using $C_{1,o}$-p $C_{2,o}$-p finite-Cs by auto

— We go down to the ground world.
— Finding the falsifying ground instance $C_1'$ of $C_{1,o} \cdot \lambda x . ("1" \oplus x)$, and proving properties about it:

— $C_1'$ is falsified by $B \oplus [\text{True}]$:
from $C_{1,p}$ obtain $C_1'$ where $C_{1}'$-p: ground$_{ls}$ $C_1'$ \land \text{instance-of$_{ls}$} $C_1'$ \land falsifies$_g$ $?B_1 C_{1}'$ by metis

have $\neg$falsifies, $B C_{1,o}$ using $C_{1,o}$-p $b$-p clo unfolding closed-tree-def by metis
then have $\neg$falsifies, $B ?C_1$ using $\text{std}_1$-falsifies using prod.exhaust-set by blast

— $C_1'$ is not falsified by $B$:
then have $l-B$: $\neg$falsifies, $B C_{1}'$ using $C_{1}'$-p by auto

— $C_1'$ contains a literal $l_1$ that is falsified by $B \oplus [\text{True}]$, but not $B$:
from $C_{1}'$-p $l-B$ obtain $l_1$ where $l_1$-p: $l_1 \in C_{1}' \land \text{falsifies}_l (B \oplus [\text{True}])$ $l_1 \land
\[ \neg (\text{falsifies}_1 B l_1) \text{ by auto} \]

\text{let } ?i = \text{nat-of-fatom (get-atom } l_1) \]

— \text{l}_1 \text{ is of course ground;}
\text{have ground-}l_1: \text{ground}_1 l_1 \text{ using } C_1' - p l_1 - p \text{ by auto}

\text{from } l_1 - p \text{ have } \neg (\neg i < \text{length } B \land B ! ?i = (\neg \text{sign } l_1)) \text{ using ground-}l_1

\text{unfolding falsifies1-def by meson}

\text{then have } \neg (\neg i < \text{length } B \land (B @ [\text{True}]) ! ?i = (\neg \text{sign } l_1)) \text{ by (metis nth-append)} — \text{Not falsified by } B.

\text{moreover}
\text{from } l_1 - p \text{ have } ?i < \text{length } (B @ [\text{True}]) \land (B @ [\text{True}]) ! ?i = (\neg \text{sign } l_1)

\text{unfolding falsifies1-def by meson}

\text{ultimately}
\text{have } l_1 - \text{sign-no: } ?i = \text{length } B \land (B @ [\text{True}]) ! ?i = (\neg \text{sign } l_1) \text{ by auto}

— \text{l}_1 \text{ is negative;}
\text{from } l_1 - \text{sign-no have } l_1 - \text{sign: sign } l_1 = \text{False by auto}
\text{from } l_1 - \text{sign-no have } l_1 - \text{no: nat-of-fatom (get-atom } l_1) = \text{length } B \text{ by auto}

— \text{All the other literals in } C_1' \text{ must be falsified by } B, \text{since they are falsified by } B @ [\text{True}], \text{but not } l_1.

\text{from } C_1' - p \text{ l}_1 - \text{no } l_1 - p \text{ have } B - C_1' l_1: \text{falsifies}_g B (C_1' - \{l_1\})

\text{using other-falsified by blast}

— We do the same exercise for \text{C}_2 o \ \rho_s (\lambda x. \in \{2'' \@ x\}), \text{C}_2', \text{B @ [False]}, \text{l}_2:
\text{from } C_2 - p \text{ obtain } C_2' \text{ where } C_2' - p: \text{ground}_1 s \text{ C}_2' \land \text{instance-of}_1 s \text{ C}_2' ?C_2

\text{and falsifies}_g ?B_2 C_2' \text{ by mesis}

\text{have } \neg \text{falsifies}_g B \text{ C}_2 o \text{ using } C_2 o - p b - p \text{ clo unfolding closed-tree-def by mesis}
\text{then have } \neg \text{falsifies}_g B ?C_2 \text{ using std2-falsifies using prod.exhaust-set by blast}

\text{then have } l - B: \neg \text{falsifies}_g B \text{ C}_2' \text{ using } C_2' - p \text{ by auto}

— \text{C}_2' \text{ contains a literal } l_2 \text{ that is falsified by } B @ [\text{False}], \text{but not } B:
\text{from } C_2' - p \text{ l-B obtain } l_2 \text{ where } l_2 - p: l_2 \in C_2' \land \text{falsifies}_1 (B @ [\text{False}]) l_2 \land

\neg \text{falsifies}_1 B l_2 \text{ by auto}
\text{let } ?i = \text{nat-of-fatom (get-atom } l_2) \]

\text{have ground-}l_2: \text{ground}_1 l_2 \text{ using } C_2' - p l_2 - p \text{ by auto}

\text{from } l_2 - p \text{ have } \neg (\neg i < \text{length } B \land B ! ?i = (\neg \text{sign } l_2)) \text{ using ground-}l_2

\text{unfolding falsifies1-def by meson}

\text{then have } \neg (\neg i < \text{length } B \land (B @ [\text{False}]) ! ?i = (\neg \text{sign } l_2)) \text{ by (metis nth-append)} — \text{Not falsified by } B.

\text{moreover}
\text{from } l_2 - p \text{ have } ?i < \text{length } (B @ [\text{False}]) \land (B @ [\text{False}]) ! ?i = (\neg \text{sign } l_2)

\text{unfolding falsifies1-def by meson}

\text{ultimately}

61
have \( l_2 \)-sign-no: \( \forall i = \text{length} \ B \land (B @ [\text{False}]) ! \forall i = \lnot \text{sign} \ l_2 \) by auto

— \( l_2 \) is negative:
  from \( l_2 \)-sign-no have \( l_2 \)-sign: \( \text{sign} \ l_2 = \text{True} \) by auto
  from \( l_2 \)-sign-no have \( l_2 \)-no: \( \text{nat-of-fatom} (\text{get-atom} \ l_2) = \text{length} \ B \) by auto

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

— Proving some properties about \( C_1' \) and \( C_2' \), \( l_1 \) and \( l_2 \), as well as the resolvent of \( C_1' \) and \( C_2' \):
have \( l_2 \text{cisl}_1 \): \( l_2^c = l_1 \)
proof
  from \( l_1 \)-no \( l_2 \)-no ground-\( l_1 \) ground-\( l_2 \) have get-atom \( l_1 \) = get-atom \( l_2 \)
  using nat-of-fatom-bij ground-\( l_1 \)-ground-\( l_2 \)
  unfolding bij-betw-def inj-on-def by metis
  then show \( l_2^c = l_1 \) using \( l_1 \)-sign \( l_2 \)-sign using sign-comp-atom by metis
qed

have applicable \( C_1' \) \( C_2' \) \{\( l_1 \)\} \{\( l_2 \)\} Resolution.e unfoldng applicable-def
  using \( l_1 \)-p \( l_2 \)-p \( C_1' \)-p ground-\( l_1 \)-vars \( l_2 \) \text{cisl}_1 \) empty-\( \text{comp}_2 \) unfoldng \( \text{mgv}_1 \)-def
  unfoldng \( \text{unifier}_1 \)-def by auto
  — Lifting to get a resolvent of \( C_1 \circ \gamma_1 \)s (\( \lambda x. \epsilon ("1" \ @ \ x) \)) and \( C_2 \circ \gamma_2 \)s (\( \lambda x. \epsilon ("2" \ @ \ x) \)):
  then obtain \( L_1 \) \( L_2 \) \( \tau \) where \( L_1 L_2 \tau \)-p: applicable \( ?C_1 \) \( ?C_2 \) \( L_1 \) \( L_2 \) \( \tau \) \land \( \text{instance-of}_1 \) (resolution \( C_1' \) \( C_2' \) \{\( l_1 \)\} \{\( l_2 \)\} Resolution.e) (resolution \( ?C_1 \) \( ?C_2 \) \( L_1 \) \( L_2 \) \( \tau \))
  using std-apart-apart \( C_1' \)-p \( C_2' \)-p lifting[\( \text{of} \) ?\( C_1 \) ?\( C_2 \) \( C_1' \) \( C_2' \) \{\( l_1 \)\} \{\( l_2 \)\}] Resolution.e] 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: \( C = \text{resolution} \) ?\( C_1 \) ?\( C_2 \) \( L_1 \) \( L_2 \) \( \tau \) by auto
obtain \( \text{CsNext} \) where \( \text{CsNext}-p: \text{CsNext} = \text{Cs} \cup \{ ?C_1 , ?C_2 , C \} \) by auto
obtain \( \text{T'} \) where \( \text{T''}-p: \text{T'} = \text{delete} \ B \ T \) by auto
  — Here we delete the two branch children \( B @ [\text{True}] \) and \( B @ [\text{False}] \) of \( B \).

— Our new clause is falsified by the branch \( B \) of our new tree:
  have \( \text{falsifies}_g \ B ((C_1' - \{l_1\}) \cup (C_2' - \{l_2\})) \) using \( B \cdot C_1' l_1 \) \( B \cdot C_2' l_2 \) by cases auto
  then have \( \text{falsifies}_g \ B (\text{resolution} \ C_1' \ C_2' \{l_1\} \{l_2\}) \) Resolution.e unfoldng resolution-def empty-\( \text{subls} \) by auto
  then have \( \text{falsifies}_{-C}: \text{falsifies}_{-C} \ B \ C \) using \( \text{C}-p \ L_1 L_2 \tau \)-p by auto

have \( \text{T''}-\text{smaller}: \text{treesize} \ T'' < \text{treesize} \ T \) using \( \text{treezise-delete} \ T''-p \ b-p \) by auto
have \( T''\)-bran: anybranch \( T'' \) (\( \lambda b. \text{closed-branch } b T'' \) \( \text{CsNext} \))

proof (rule allI; rule impI)
  fix \( b \)
  assume \( br: \text{branch } b T'' \)
  from \( br \) have \( b = B \lor \text{branch } b T \) using branch-delete \( T''\)-p by auto
  then show \( \text{closed-branch } b T'' \) \( \text{CsNext} \)
    proof
      assume \( b = B \)
      then show \( \text{closed-branch } b T'' \) \( \text{CsNext} \)
        using falsifies-C \( b \) \( \text{CsNext-p} \) by(auto)
  next
    assume \( \text{branch } b T \)
    then show \( \text{closed-branch } b T'' \) \( \text{CsNext} \)
      using clo \( b \) \( T''\)-p \( \text{CsNext-p} \)
      unfolding closed-tree-def by(auto)
  qed
  qed

then have \( T''\)-bran2: anybranch \( T'' \) (\( \lambda b. \text{falsifies}_{cs} b \) \( \text{CsNext} \)) 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: \( T' = \text{cutoff} (\lambda G. \text{falsifies}_{cs} G \) \( \text{CsNext} \)) \[ \[] T'' \) by auto
have \( T'\)-smaller: treesize \( T' < \) treesize \( T \) using treesize-cutoff[\( \lambda G. \text{falsifies}_{cs} G \) \( \text{CsNext} \) \[ \[] \) \[ \] \( T'' \)-smaller unfolding \( T''\)-p by(auto)

from \( T''\)-bran2 have anybranch \( T' \) (\( \lambda b. \text{falsifies}_{cs} b \) \( \text{CsNext} \)) using cutoff-branch[\( \lambda b. \text{falsifies}_{cs} b \) \( \text{CsNext} \) \( T''\)-p by(auto)
  then have \( T'\)-bran: anybranch \( T' \) (\( \lambda b. \text{closed-branch } b T' \) \( \text{CsNext} \)) by(auto)
  have \( T'\)-intr: anyinternal \( T' \) (\( \lambda p. \neg \text{falsifies}_{cs} p \) \( \text{CsNext} \)) using \( T'\)-p cutoff-internal[\( \lambda G. \text{falsifies}_{cs} G \) \( \text{CsNext} \) \[ \[] \) \( T''\)-bran2 by blast
  have \( T'\)-closed: closed-tree \( T' \) \( \text{CsNext} \) using \( T'\)-bran \( T'\)-intr unfolding closed-tree-def by(auto)
  have finite-CsNext: \( \forall C \in \text{CsNext}. \text{finite } C \) unfolding \( \text{CsNext-p} \) by(auto)

— By induction hypothesis we get a resolution derivation of \( \{\} \) from our new
  clausal form:

from \( T''\)-smaller \( T'\)-closed have \( \exists Cs''. \text{resolution-deriv } \text{CsNext } Cs'' \land \{\} \in Cs'' \) using ih[\( \lambda T' \) \( \text{CsNext} \) \[ \] \[ \] 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 std1-renames standardize-apart \( C_{1-o-p} \) by (metis Un-insert-right)
    moreover
      have resolution-step \( Cs \cup \{?C_1\}\) \( \{Cs \cup \{?C_1\} \cup \{?C_2\}\} \) using std2-renames[\( \lambda C_{2-o} \) standardize-apart[\( \lambda C_{2-o} \) \( ?C_2 \)] \( C_{2-o-p} \) by(auto)
      then have resolution-step \( Cs \cup \{?C_1\}\) \( \{Cs \cup \{?C_1,?C_2\}\} \) by (simp add: insert-commute)
moreover

then have resolution-step \((Cs \cup \{?C_1, ?C_2\}) \ (Cs \cup \{?C_1, ?C_2\} \cup \{C\})\)

using \(L_1 L_2\)\(\cdot\)-p resolution-rule[ of \(?C_1\ Cs \cup \{?C_1, ?C_2\} ?C_2 \) \(L_1 L_2 \tau\)] using

\(\mbox{C-p by auto}\)

then have resolution-step \((Cs \cup \{?C_1, ?C_2\}) \ CsNext\) using \(CsNext\)-p by

\((\mbox{simp add: Un-commute})\)

ultimately

have resolution-deriv Cs CsNext unfolding resolution-deriv-def by auto

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 CsNext unfolding resolution-deriv-def 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: \(\forall (F::\mbox{hterm} \ \mbox{fun-denot}) \ (G::\mbox{hterm} \ \mbox{pred-denot}) \ .\ \neg\ \mbox{eval}\_cs\ F G Cs\)

shows \(\exists Cs'.\ \mbox{resolution-deriv} Cs Cs' \land \{\} \in Cs'\)

proof —

from unsat have \(\forall (G::\mbox{hterm} \ \mbox{pred-denot}) \ .\ \neg\ \mbox{eval}\_cs\ HFun G Cs\) by auto

then obtain \(T\) where closed-tree \(T\) Cs using herbrand assms by blast

then show \(\exists Cs'.\ \mbox{resolution-deriv} Cs Cs' \land \{\} \in Cs'\) using completeness’ assms by auto

qed

definition E-conv :: \((\mbox{\char39}a \Rightarrow \mbox{\char39}b)\) \Rightarrow \mbox{\char39}a \ \mbox{var-denot} \Rightarrow \mbox{\char39}b \ \mbox{var-denot} \ where

\(E-conv\ b-of-a\ E \equiv \lambda x.\ (b-of-a\ (E\ x))\)

definition F-conv :: \((\mbox{\char39}a \Rightarrow \mbox{\char39}b)\) \Rightarrow \mbox{\char39}a \ \mbox{fun-denot} \Rightarrow \mbox{\char39}b \ \mbox{fun-denot} \ where

\(F-conv\ b-of-a\ F \equiv \lambda f\ bs.\ b-of-a\ (F\ f\ (\mbox{map}\ \mbox{inv}\ b-of-a\ bs))\)

definition G-conv :: \((\mbox{\char39}a \Rightarrow \mbox{\char39}b)\) \Rightarrow \mbox{\char39}a \ \mbox{pred-denot} \Rightarrow \mbox{\char39}b \ \mbox{pred-denot} \ where

\(G-conv\ b-of-a\ G \equiv \lambda p\ bs.\ (G\ p\ (\mbox{map}\ \mbox{inv}\ b-of-a\ bs))\)

lemma eval\_t-bij:

assumes bij \((b-of-a::\mbox{\char39}a \Rightarrow \mbox{\char39}b)\)

shows \(\mbox{eval}_t\ (E-conv\ b-of-a\ E)\ (F-conv\ b-of-a\ F)\ t = b-of-a\ (\mbox{eval}_t\ E\ F\ t)\)

proof (induction t)

case (Fun f ts)

then have \(\mbox{map}\ (\mbox{inv}\ b-of-a \circ \mbox{eval}_t\ (E-conv\ b-of-a\ E)\ (F-conv\ b-of-a\ F))\ ts = \mbox{eval}_t\ E\ F\ ts\)

unfolding E-conv-def F-conv-def

using assms bij-is-inj by fastforce

then have \(b-of-a\ (F\ f\ (\mbox{map}\ (\mbox{inv}\ b-of-a \circ \mbox{eval}_t\ (E-conv\ b-of-a\ E)\ ((F-conv\ b-of-a\ F))\ ts)) = b-of-a\ (F\ f\ (\mbox{eval}_t\ E\ F\ ts))\) by metis

then show ?case using assms unfolding E-conv-def F-conv-def by auto

64
next  
case (Var x)  
then show ?case using assms unfolding E-conv-def by auto  
qed

lemma eval-ts-bij:  
assumes bij (b-of-a::'a ⇒ 'b)  
shows G-conv b-of-a G p (eval-ts (E-conv b-of-a E) (F-conv b-of-a F) ts) = G p (eval-ts E F ts)  
using assms using eval-t bij  
proof –  
have map (inv b-of-a o eval-t (E-conv b-of-a E) (F-conv b-of-a F)) ts = eval-ts E F ts  
using eval-t bij assms bij-is-inj by fastforce  
then show ?thesis  
by (metis (no-types) G-conv-def map-map)  
qed

lemma eval-l-bij:  
assumes bij (b-of-a::'a ⇒ 'b)  
shows eval-l (E-conv b-of-a E) (F-conv b-of-a F) (G-conv b-of-a G) l = eval-l E F G l  
using assms eval-ts-bij  
proof (cases l)  
case (Pos p ts)  
then show ?thesis  
by (simp add: eval-ts-bij assms)  
next  
case (Neg p ts)  
then show ?thesis  
by (simp add: eval-ts-bij assms)  
qed

lemma eval-c-bij:  
assumes bij (b-of-a::'a ⇒ 'b)  
shows eval-c (F-conv b-of-a F) (G-conv b-of-a G) C = eval-c F G C  
proof –  
{  
fix E :: char list ⇒ 'b  
assume bij-b-of-a: bij b-of-a  
assume C-sat: ∀ E :: char list ⇒ 'a. ∃ l∈C. eval-l E F G l  
have E-p: E = E-conv b-of-a (E-conv (inv b-of-a) E)  
umfolding E-conv-def using bij-b-of-a  
using bij-beta-into-right by fastforce  
have ∃ l∈C. eval-l (E-conv b-of-a (E-cone (inv b-of-a) E)) (F-conv b-of-a F) (G-conv b-of-a G) l  
using eval-l bij bij-b-of-a C-sat by blast  
then have ∃ l∈C. eval-l E (F-conv b-of-a F) (G-conv b-of-a G) l using E-p  
}
by auto
}
then show ?thesis
  
  by (meson eval1-bij assms eval_c-def)
qed

lemma eval_c-bij:
  assumes bij (b-of-a::'a ⇒ 'b)
  shows eval_c (F-conv b-of-a F) (G-conv b-of-a G) Cs ←→ eval_c F G Cs
  by (meson eval_c-bij assms eval_c-def)

lemma countably-inf-bij:
  assumes inf-a-uni: infinite (UNIV :: ('a ::countable) set)
  assumes inf-b-uni: infinite (UNIV :: ('b ::countable) set)
  shows ∃ b-of-a :: 'a ⇒ 'b. bij b-of-a
proof
  let ?S = UNIV :: ('a ::countable) set
  have countable ?S by auto
  moreover
  have infinite ?S using inf-a-uni by auto
  ultimately
  obtain nat-of-a where QWER: bij (nat-of-a :: 'a ⇒ nat) using countableE-infinite[of ?S] by blast

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

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

  have bij-nat-of-b: ∀ n. nat-of-b (inv nat-of-b n) = n
    using TYUI bij-betw-inv-into-right by fastforce
  have ∀ a. inv nat-of-a (nat-of-a a) = a
    by (meson QWER UNIV-I bij-betw-inv-into-left)
  then have inj (λ a. inv nat-of-b (nat-of-a a))
    using bij-nat-of-b injI by (metis (no-types))
  moreover
  have range (λ a. inv nat-of-b (nat-of-a a)) = UNIV
    by (meson QWER TYUI bij-def image_image inj_imp_surj_inv)
  ultimately
  have bij ?b-of-a
    unfolding bij-def by auto
  then show ?thesis by auto
qed
lemma infinite-hterms: infinite (UNIV :: hterm set)
proof -
let ?diago = λn. HFun (string-of-nat n) []
let ?undiago = λa. nat-of-string (case a of HFun f ts ⇒ f)
have ∀ n. ?undiago (?diago n) = n using nat-of-string-string-of-nat by auto
moreover
have ∀ n. ?diago n ∈ UNIV by auto
ultimately show infinite (UNIV :: hterm set) using infinity[of ?undiago ?diago UNIV] by simp
qed

theorem completeness-countable:
  assumes inf-uni: infinite (UNIV :: ('u :: countable) set)
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  assumes unsat: ∀ (F::'u fun-denot) (G::'u pred-denot). ¬evalcs F G Cs
  shows ∃ Cs'. resolution-deriv Cs Cs' ∧ {} ∈ Cs'
proof -
  have ∀ (F::hterm fun-denot) (G::hterm pred-denot). ¬evalcs F G Cs
proof (rule; rule)
    fix F :: hterm fun-denot
    fix G :: hterm pred-denot
  obtain u-of-hterm :: hterm ⇒ 'u where p-u-of-hterm: bij u-of-hterm
    using countably-inf-bij inf-uni infinite-hterms by auto
  let ?F = F-conv u-of-hterm F
  let ?G = G-conv u-of-hterm G
  have ¬ evalcs ?F ?G Cs using unsat by auto
  then show ¬ evalcs F G Cs using evalcs-bij using p-u-of-hterm by auto
  qed
  then show ∃ Cs'. resolution-deriv Cs Cs' ∧ {} ∈ Cs'
  using finite-cs completeness
  by auto
  qed

theorem completeness-nat:
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  assumes unsat: ∀ (F::nat fun-denot) (G::nat pred-denot). ¬evalcs F G Cs
  shows ∃ Cs'. resolution-deriv Cs Cs' ∧ {} ∈ Cs'
  using assms completeness-countable by blast

end — unification locale

end

18 Examples

theory Examples imports Resolution begin
value Var "x"
value Fun "one" []
value Fun "mul" [Var "y", Var "y"]
value Fun "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []]

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

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

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

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

lemma eval 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

lemma eval E nat F nat G nat (Pos "greater" [Var "x", Var "y"] ) = True
by auto
lemma eval E nat F nat G nat (Neg "less" [Var "x", Var "y"] ) = True
by auto
lemma eval E nat F nat G nat (Pos "less" [Var "x", Var "y"] ) = False
by auto

lemma eval $E_{nat}$ $F_{nat}$ $G_{nat}$
$\begin{array}{l}
(\text{Pos } \text{"equals"} \\
[\text{Fun } \text{"add"} [\text{Fun } \text{"mul"} [\text{Var } \text{"y"}, \text{Var } \text{"y"}], \text{Fun } \text{"one"}]] \\
, \text{Var } \text{"x"}] \\
) = \text{ True}
\end{array}$
by auto

definition $PP :: \text{fterm literal where}$
$PP = \text{Pos } \text{"P"} [\text{Fun } \text{"c"} []]$

definition $PQ :: \text{fterm literal where}$
$PQ = \text{Pos } \text{"Q"} [\text{Fun } \text{"d"} []]$

definition $NP :: \text{fterm literal where}$
$NP = \text{Neg } \text{"P"} [\text{Fun } \text{"c"} []]$

definition $NQ :: \text{fterm literal where}$
$NQ = \text{Neg } \text{"Q"} [\text{Fun } \text{"d"} []]$

theorem empty-mgu:
assumes unifier$_{1s} \in L$
shows mgu$_{1s} \in L$
using assms unfolding unifier$_{1s}$-def mgu$_{1s}$-def apply auto
apply (rule-tac $x=a$ in exI)
using empty-comp1 empty-comp2 apply auto
done

theorem unifier-single: unifier$_{1s} \sigma \{l\}$
unfolding unifier$_{1s}$-def by auto

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

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

have resolution-step
$\{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}$
$\{\{NP,PQ\},\{NQ\},\{PP,PQ\}\} \cup \{\{NP\}\}$
apply (rule resolution-rule' of $\{NP,PQ\} - \{NQ\} \{PQ\} \{NQ\} \epsilon$)
unfolding applicable-def vars$_{1s}$-def vars$_{1}$-def
NQ-def NP-def PQ-def PP-def resolution-def
using unifier-single empty-mgu using empty-subls
apply auto
done
then have resolution-step
\{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}
\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\}\}
by (simp add: insert-commute)
moreover
have resolution-step
\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\}\}
\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\},\{PP\}\}
apply (rule resolution-rule [of \{NQ\} - \{PP,PQ\} \{NP\} \{PP\} ∈])
unfolding applicable-def vars1-def vars2-def
NQ-def NP-def PQ-def PP-def resolution-def
using unifier-single empty-mgu empty-subls apply auto
done
moreover
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 [of \{NP\} - \{PP\} \{NP\} \{PP\} ∈])
unfolding applicable-def vars1-def vars2-def
NQ-def NP-def PQ-def PP-def resolution-def
using unifier-single empty-mgu apply auto
done
ultimately
have resolution-deriv \{\{NP,PQ\},\{NQ\},\{PP,PQ\}\}
\{\{NP,PQ\},\{NQ\},\{PP,PQ\},\{NP\},\{PP\},\{\}\}
unfolding resolution-deriv-def by auto
then show ?thesis by auto
qed

definition Pa :: fterm literal where
Pa = Pos "a" []

definition Na :: fterm literal where
Na = Neg "a" []

definition Pb :: fterm literal where
Pb = Pos "b" []
definition Nb :: fterm literal where
Nb = Neg "b" []

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

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

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

definition Nax :: fterm literal where
Nax = Neg "a" [Var "x"

definition mguPaaPax :: substitution where
mguPaaPax = (λx. if x = "x" then Fun "a" [] else Var x)

lemma mguPaaPax-mgu : mgu ls mguPaaPax {Paa,Pax}
proof
  let ?σ = λx. if x = "x" then Fun "a" [] else Var x
  have a: unifier ls (λx. if x = "x" then Fun "a" [] else Var x) {Paa,Pax} unfolding Paa-def Pax-def
  proof (rule:rule)
    fix u
    assume unifier ls u {Paa,Pax}
    then have uu u "x" = Fun "a" [] unfolding unifier ls-def Paa-def Pax-def
  proof (rule:rule)
    fix x
    assume x="x"
    moreover
    have (/?σ·u) "x" = Fun "a" [] unfolding composition-def by auto
    ultimately have (/?σ·u) x = u x using uu u by auto
  }
  moreover
  {assume x ≠ "x"
    then have (/?σ·u) x = (ε x) x u unfolding composition-def by auto
    then have (/?σ·u) x = u x by auto
  }
  ultimately show (/?σ·u) x = u x by auto
qed
then have ∃i. ?σ·i = u by auto
then show ∃i. u = ?σ·i by auto
qed

from a b show thesis unfolding mguPaaPax-def unfolding Na-def by auto

qed

lemma resolution-example2:
resolution-deriv \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}
\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\},\{\}\}\n
proof -
have resolution-step \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}
(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\} \cup \{\{Na,Pb\}\})
apply (rule resolution-rule[of \{Pax\} - \{Na,Pb,Naa\} \{Pa\} \{Naa\} mguPaaPax ])
using mguPaaPax-mgu unfolding applicable-def varsP-def varsA-def
Nb-def Na-def Pax-def Pa-def Pb-def Naa-def Paa-def mguPaaPax-def

resolution-def
apply auto
apply (rule-tac x=Na in image-eqI)
unfolding Na-def apply auto
apply (rule-tac x=Pb in image-eqI)
unfolding Pb-def apply auto
done

then have resolution-step \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}
(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\} )
by (simp add: insert-commute)
moreover
have resolution-step \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\}
(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\} \cup \{\{Na\}\})
apply (rule resolution-rule[of \{Nb,Na\} - \{Na,Pb\} \{Nb\} \{Pb\} \in])
unfolding applicable-def varsP-def varsA-def
Pb-def Nb-def Na-def PP-def resolution-def
using unifier-single empty-mgu apply auto
done

then have resolution-step \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\}
(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\} )
by (simp add: insert-commute)
moreover
have resolution-step \{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\}
(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\} \cup \{\{\}\} )
apply (rule resolution-rule[of \{Na\} - \{Pa\} \{Na\} \{Pa\} \in])
unfolding applicable-def varsP-def varsA-def
Pa-def Nb-def Na-def PP-def resolution-def
using unifier-single empty-mgu apply auto
done

72
then have resolution-step
   \{(\{Nb,Na\},\{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\},\{Na,Pb\},\{Na\},\{\}\)
   unfolding resolution-deriv-def by auto
then show ?thesis by auto
qed

lemma resolution-example1-sem: \\lnot evalcs F G \{(\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}
using resolution-example1 derivation-sound-refute by auto

lemma resolution-example2-sem: \\lnot evalcs F G \{(\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}
using resolution-example2 derivation-sound-refute by auto

end

19 The Unification Theorem

theory Unification-Theorem imports
   First-Order-Terms.Unification Resolution
begin

definition set-to-list :: 'a set \Rightarrow 'a list where
definition set-to-list = inv set

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

fun iterm-to-fterm :: (fun-sym, var-sym) term \Rightarrow fterm where
   iterm-to-fterm (Term.Var x) = Var x
| iterm-to-fterm (Term.Fun f ts) = Fun f (map iterm-to-fterm ts)

fun fterm-to-iterm :: fterm \Rightarrow (fun-sym, var-sym) term where
   fterm-to-iterm (Var x) = Term.Var x
| fterm-to-iterm (Fun f ts) = Term.Fun f (map fterm-to-iterm ts)

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

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

abbreviation (input) fsup-to-isub :: substitution => (fun-sym, var-sym) subst where
  fsup-to-isub σ = λx. fterm-to-iterm (σ x)

abbreviation (input) isub-to-fsup :: (fun-sym, var-sym) subst => substitution where
  isub-to-fsup σ = λx. iterm-to-fterm (σ x)

lemma iterm-to-fterm-subt: (iterm-to-fterm t1) · t σ = iterm-to-fterm (t1 · (λx. fterm-to-iterm (σ x)))
by (induction t1) auto

lemma unifiert-unifiers:
  assumes unifier ts σ ts
  shows fsup-to-isub σ ∈ unifiers (fterm-to-iterm ' ts × fterm-to-iterm ' ts)
proof
  have ∀ t1 ∈ fterm-to-iterm ' ts. ∀ t2 ∈ fterm-to-iterm ' ts. t1 · (fsup-to-isub σ) = t2 · (fsup-to-isub σ)
  proof (rule ballI; rule ballI)
    fix t1 t2
    assume t1-p: t1 ∈ fterm-to-iterm ' ts assume t2-p: t2 ∈ fterm-to-iterm ' ts
    from t1-p t2-p have iterm-to-fterm t1 ∈ ts ∧ iterm-to-fterm t2 ∈ ts by auto
    then have (iterm-to-fterm t1) · t σ = (iterm-to-fterm t2) · t σ using assms unfolding unifier ts-def by auto
    then have iterm-to-fterm (t1 · fsup-to-isub σ) = iterm-to-fterm (t2 · fsub-to-isub σ) using iterm-to-fterm-subt by auto
    then have fterm-to-iterm (iterm-to-fterm (t1 · fsup-to-isub σ)) = fterm-to-iterm (iterm-to-fterm (t2 · fsup-to-isub σ)) by auto
    then show t1 · fsup-to-isub σ = t2 · fsub-to-isub σ using fterm-to-iterm-cancel by auto
    qed
  then have ∀ p ∈ fterm-to-iterm ' ts × fterm-to-iterm ' ts. fst p · fsub-to-isub σ = snd p · fsup-to-isub σ by (metis mem-Times-iff)
  then show ?thesis unfolding unifiers-def by blast
qed

abbreviation (input) get-mgut :: fterm list => substitution option where
  get-mgut ts = map-option (isub-to-fsup o subst-of) (unify (List.product (map fterm-to-iterm ts) (map fterm-to-iterm ts)) [])

lemma unify-unification:
  assumes σ ∈ unifiers (set E)
  shows ∃ϑ. is-imgu ϑ (set E)
proof
  from assms have ∃cs. unify E [] = Some cs using unify-complete by auto
then show \( \text{thesis} \) using unify-sound by auto

qed

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

by (induction \( t1 \)) auto

lemma \text{unifiers-unifiert}:

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

shows \( \text{unifier}_ts (\text{isub-to-fsub} \sigma) ts \)

proof (cases \( ts = \{\} \))

assume \( ts = \{\} \)

then show \( \text{unifier}_ts (\text{isub-to-fsub} \sigma) ts \)

unfolding \text{unifier}_ts-def by auto

next

assume \( ts \neq \{\} \)

then obtain \( t' \) where \( t' \in ts \) by auto

have \( \forall t_1 \in ts. \forall t_2 \in ts. t_1 \cdot (\text{isub-to-fsub} \sigma) = t_2 \cdot (\text{isub-to-fsub} \sigma) \)

proof (rule ballI ; rule ballI)

fix \( t_1 \) \( t_2 \)

assume \( t_1 \in ts t_2 \in ts \)

then have \( fterm-to-iterm t_1 \in fterm-to-iterm ' ts \)

(\( fterm-to-iterm t_2 \in fterm-to-iterm ' ts \) \)

by auto

then have \( (fterm-to-iterm t_1, fterm-to-iterm t_2) \in (fterm-to-iterm ' ts \times fterm-to-iterm ' ts) \) \)

by auto

then have \( \{\} \)

then obtain \( t' \) where \( t' \in ts \) by auto

then show \( \text{unifier}_ts (\text{isub-to-fsub} \sigma) ts \)

unfolding \text{unifier}_ts-def by metis blast

qed

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

unfolding \text{composition-def subst-compose-def}

proof

fix \( x \)

show \( \vartheta x \cdot i = fterm-to-iterm (iterm-to-fterm (\vartheta x) \cdot (\lambda x. \text{iterm-to-fterm} (i x))) \)

using \text{iterm-to-fterm-subt by auto}

qed

lemma \text{is-mgu-mgu}_{ts}:
assumes finite ts
assumes is-imgu \vartheta \ (fterm-to-iterm \ ' ts \times fterm-to-iterm \ ' ts)
shows mgu_{ts} (isub-to-fsub \ \vartheta) \ ts

proof
from assms have unifier_{ts} (isub-to-fsub \ \vartheta) \ ts unfolding is-imgu-def using
unifiers-unifier by auto
moreover have \forall u. \ unifier_{ts} u \ ts \rightarrow (\exists i. \ u = (isub-to-fsub \ \vartheta) \cdot i)
proof (rule allI; rule impI)
fix u
assume unifier_{ts} u \ ts
then have fsub-to-isub u \in unifiers (fterm-to-iterm \ ' ts \times fterm-to-iterm \ ' ts)
using unifier-unifiers by auto
then have \exists i. \ fsub-to-isub u = \vartheta \circ s \cdot i using assms unfolding is-imgu-def
by auto
then obtain i where fsub-to-isub u = \vartheta \circ s \cdot i by auto
then have fsub-to-isub u = fsub-to-isub (isub-to-fsub \vartheta \cdot isub-to-fsub i) using
icomp-fcomp by auto
then have isub-to-fsub (fsub-to-isub u) = isub-to-fsub (fsub-to-isub (isub-to-fsub
\vartheta \cdot isub-to-fsub i)) by metis
then have u = isub-to-fsub \vartheta \cdot isub-to-fsub i \ by auto
then show \exists i. \ u = isub-to-fsub \vartheta \cdot i \ by metis
qed
ultimately show \exists \vartheta. \ mgu_{ts} \vartheta \ ts unfolding mgu_{ts}-def by auto
qed

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

fun literal-to-term :: fterm literal \Rightarrow fterm where
literal-to-term (Pos p ts) = Fun "Pos" [Fun p ts]
| literal-to-term (Neg p ts) = Fun "Neg" [Fun p ts]

fun term-to-literal :: fterm \Rightarrow fterm literal where
term-to-literal (Fun s [Fun p ts]) = (if s="Pos" then Pos else Neg) p ts

lemma term-to-literal-cancel(simp): term-to-literal (literal-to-term l) = l
by (cases l) auto

76
lemma `literal-to-term-sub`: \[ \text{literal-to-term} \ (l \cdot l \sigma) = \text{literal-to-term} \ (l) \cdot l \sigma \]
by (induction \( l \)) auto

lemma `unifier_t_s-unifier_t_s`:
assumes `unifier_t_s \ sigma L`
shows `unifier_t_s \ sigma (\text{literal-to-term} \ \cdot \ L)`
proof –
from `assms` obtain \( l' \) where \( \forall l \in L. \ l \cdot l \sigma = l' \) unfolding `unifier_t_s-def` by auto
then have \( \forall l \in L. \ \text{literal-to-term} \ (l \cdot l \sigma) = \text{literal-to-term} \ (l') \) by auto
then have \( \forall l \in L. \ (\text{literal-to-term} \ l) \cdot l \sigma = \text{literal-to-term} \ l' \) using `literal-to-term-sub` by auto
then have \( \forall l \in \text{literal-to-term} \ \cdot \ L. \ t \cdot t \sigma = \text{literal-to-term} \ l' \) by auto
then show \( ?thesis \) unfolding `unifier_t_s-def` by auto
qed

lemma `unifier_t_s-unifier_t_s`:
assumes `unifier_t_s \ sigma (\text{literal-to-term} \ \cdot \ L)`
shows `unifier_t_s \ sigma L`
proof –
from `assms` obtain \( t' \) where \( \forall t \in \text{literal-to-term} \ \cdot \ L. \ t \cdot t \sigma = t' \) unfolding `unifier_t_s-def` by auto
then have \( \forall t \in \text{literal-to-term} \ \cdot \ L. \ \text{term-to-literal} \ (t \cdot t \sigma) = \text{term-to-literal} \ t' \) by auto
then have \( \forall t \in L. \ \text{term-to-literal} \ ((\text{literal-to-term} \ l) \cdot l \sigma) = \text{term-to-literal} \ t' \) using `literal-to-term-sub` by auto
then have \( \forall t \in L. \ l \cdot l \sigma = \text{term-to-literal} \ t' \) by auto
then show \( ?thesis \) unfolding `unifier_t_s-def` by auto
qed

lemma `mgu_t_s-mgu_t_s`:
assumes `mgu_t_s \ var L`
shows `mgu_t_s \ var L`
proof –
from `assms` have `unifier_t_s \ var L` unfolding `mgu_t_s-def` by auto
then have `unifier_t_s \ var L` using `unifier-t_s-unifier-t_s` by auto
moreover
\{
fix \( u \)
assume `unifier_t_s \ u L`
then have `unifier_t_s \ u L` using `unifier-t_s-unifier-t_s` by auto
then have \( \exists i. \ u = \var \cdot i \) using `assms` unfolding `mgu_t_s-def` by auto
\}
ultimately show \( ?thesis \) unfolding `mgu_t_s-def` by auto
qed

77
theorem unification:
  assumes fin: finite L
  assumes uni: unifier ls σ L
  shows ∃ ϑ. mgu ls ϑ L
proof –
  from uni have unifier ls σ (literal-term L) using unifier ls-unifier ls by auto
  then obtain ϑ where mgu ls ϑ (literal-term L) using fin unification by blast
  then have mgu ls ϑ L using mgu ls-mgu ls by auto
  then show ?thesis by auto
qed

20 Instance of completeness theorem

theory Completeness-Instance imports Unification-Theorem Completeness begin
interpretation unification using unification by unfold-locales auto
thm lifting

lemma lift:
  assumes fin: finite C ∧ finite D
  assumes apart: vars ls C ∩ vars ls D = {} 
  assumes inst1: instance-of ls C' C
  assumes inst2: instance-of ls D' D
  assumes appl: applicable C' D' L' M' σ 
  shows ∃ L M τ. applicable C D L M τ ∧
           instance-of ls (resolution C' D' L' M' σ) (resolution C D L M τ)
using assms lifting by metis

thm completeness

theorem complete:
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  assumes unsat: ∀ (F::'u func-denot) (G::'u pred-denot) . ¬eval cs F G Cs
  shows ∃ Cs'. resolution-deriv Cs Cs' ∧ {} ∈ Cs'
using assms completeness by –

thm completeness-countable

theorem complete-countable:
  assumes inf-uni: infinite (UNIV :: ('u :: countable) set)
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  assumes unsat: ∀ (F::'u func-denot) (G::'u pred-denot). ¬eval cs F G Cs
  shows ∃ Cs'. resolution-deriv Cs Cs' ∧ {} ∈ Cs'
using assms completeness-countable by –
thm completeness-nat

theorem complete-nat:
  assumes finite-cs: finite Cs \forall C \in Cs, finite C
  assumes unsat: \forall (F::nat fun-denot) (G::nat pred-denot). \lnot \text{eval}_{cs} F G Cs
  shows \exists Cs': \text{resolution-deriv} Cs Cs' \land \{\} \in Cs'
  using assms completeness-nat by –

end

References


