

The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

December 17, 2016

Abstract

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

Contents

1	Terms and Literals	2
1.1	Ground	3
1.2	Auxiliary	3
1.3	Conversions	4
1.3.1	Conversions - Terms and Herbrand Terms	4
1.3.2	Conversions - Literals and Herbrand Literals	5
1.3.3	Conversions - Atoms and Herbrand Atoms	5
1.4	Enumerations	6
1.4.1	Enumerating Strings	6
1.4.2	Enumerating Herbrand Atoms	7
1.4.3	Enumerating Ground Atoms	8
2	Trees	9
2.1	Sizes	9
2.2	Paths	9
2.3	Branches	11
2.4	Internal Paths	13
2.5	Deleting Nodes	15

3	Possibly Infinite Trees	22
3.1	Infinite Paths	23
4	König’s Lemma	24
5	More Terms and Literals	25
6	Clauses	26
7	Semantics	27
7.1	Semantics of Ground Terms	28
8	Substitutions	28
8.1	The Empty Substitution	29
8.2	Substitutions and Ground Terms	30
8.3	Composition	31
8.4	Merging substitutions	33
8.5	Standardizing apart	35
9	Unifiers	36
9.1	Most General Unifiers	38
10	Resolution	39
11	Soundness	40
12	Herbrand Interpretations	43
13	Partial Interpretations	44
14	Semantic Trees	48
15	Herbrand’s Theorem	48
16	Lifting Lemma	54
17	Completeness	55
18	Examples	63

1 Terms and Literals

```
theory TermsAndLiterals imports Main  $\sim\sim$  /src/HOL/Library/Countable-Set begin
```

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

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

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

type-synonym *'t atom* = *pred-sym * 't list*

datatype *'t literal* =
sign: Pos (get-pred: pred-sym) (get-terms: 't list)
| *Neg (get-pred: pred-sym) (get-terms: 't list)*

fun *get-atom* :: *'t literal* \Rightarrow *'t atom* **where**
get-atom (Pos p ts) = (p, ts)
| *get-atom (Neg p ts) = (p, ts)*

1.1 Ground

fun *ground_t* :: *fterm* \Rightarrow *bool* **where**
ground_t (Var x) \longleftrightarrow False
| *ground_t (Fun f ts) \longleftrightarrow ($\forall t \in \text{set } ts. \text{ground}_t t$)*

abbreviation *ground_{ts}* :: *fterm list* \Rightarrow *bool* **where**
ground_{ts} ts \equiv ($\forall t \in \text{set } ts. \text{ground}_t t$)

abbreviation *ground_l* :: *fterm literal* \Rightarrow *bool* **where**
ground_l l \equiv ground_{ts} (get-terms l)

abbreviation *ground_{ls}* :: *fterm literal set* \Rightarrow *bool* **where**
ground_{ls} C \equiv ($\forall l \in C. \text{ground}_l l$)

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

lemma *ground_l-ground-fatoms*: *ground_l l \implies get-atom l \in ground-fatoms*
unfolding *ground-fatoms-def* **by** (*induction l*) *auto*

1.2 Auxiliary

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

proof —

from *inj all-tree* **have** $\forall n. n = \text{undiago } (\text{diago } n) \wedge (\text{diago } n) \in S$ **by** *auto*
then have $\forall n. \exists ds. n = \text{undiago } ds \wedge ds \in S$ **by** *auto*

then have *undiago* ‘ $S = (UNIV :: nat\ set)$ ’ by *auto*
then show $\neg finite\ S$ by (*metis finite-imageI infinite-UNIV-nat*)
qed

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

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

1.3 Conversions

1.3.1 Conversions - Terms and Herbrand Terms

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

definition *fterms-of-hterms* :: $hterm\ list \Rightarrow fterm\ list$ **where**
fterms-of-hterms $ts \equiv map\ fterm-of-hterm\ ts$

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

definition *hterms-of-fterms* :: $fterm\ list \Rightarrow hterm\ list$ **where**
hterms-of-fterms $ts \equiv map\ hterm-of-fterm\ ts$

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

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

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

lemma [*simp*]: *ground_{t,s}* $ts \implies fterms-of-hterms\ (hterms-of-fterms\ ts) = ts$
unfolding *fterms-of-hterms-def hterms-of-fterms-def* **by** (*simp add: map-idI*)

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

lemma *ground-fterms-of-hterms*: *ground_{t,s}* $(fterms-of-hterms\ ts)$
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* \Rightarrow *fterm literal* **where**
flit-of-hlit (*Pos p ts*) = *Pos p (fterms-of-hterms ts)*
| *flit-of-hlit* (*Neg p ts*) = *Neg p (fterms-of-hterms ts)*

fun *hlit-of-flit* :: *fterm literal* \Rightarrow *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*: *ground₁ (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*]: *ground₁ l \implies flit-of-hlit (hlit-of-flit l) = l*
by (*cases l*) *auto*

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

lemma *hlit-of-flit-bij*: *bij-betw hlit-of-flit {l. ground₁ l} UNIV*
unfolding *bij-betw-def*

proof

show *inj-on hlit-of-flit {l. ground₁ l}* **using** *inj-on-inverseI flit-of-hlit-hlit-of-flit*
by (*metis (mono-tags, lifting) mem-Collect-eq*)

next

have $\forall l. \exists l'. \text{ground}_1 l' \wedge l = \text{hlit-of-flit } l'$
using *ground-flit-of-hlit hlit-of-flit-flit-of-hlit* **by** *metis*
then show *hlit-of-flit ' {l. ground₁ l} = UNIV* **by** *auto*

qed

lemma *flit-of-hlit-bij*: *bij-betw flit-of-hlit UNIV {l. ground₁ l}*
unfolding *bij-betw-def inj-on-def*

proof

show $\forall x \in UNIV. \forall y \in UNIV. \text{flit-of-hlit } x = \text{flit-of-hlit } y \longrightarrow x = y$
using *ground-flit-of-hlit hlit-of-flit-flit-of-hlit* **by** *metis*

next

have $\forall l. \text{ground}_1 l \longrightarrow (l = \text{flit-of-hlit } (\text{hlit-of-flit } l))$ **using** *hlit-of-flit-flit-of-hlit*
by *auto*

then have *{l. ground₁ l} \subseteq flit-of-hlit ' UNIV* **by** *blast*
moreover

have $\forall l. \text{ground}_1 (\text{flit-of-hlit } l)$ **using** *ground-flit-of-hlit* **by** *auto*

ultimately show *flit-of-hlit ' UNIV = {l. ground₁ l}* **using** *hlit-of-flit-flit-of-hlit*
ground-flit-of-hlit **by** *auto*

qed

1.3.3 Conversions - Atoms and Herbrand Atoms

fun *fatom-of-hatom* :: *hterm atom* \Rightarrow *fterm atom* **where**

$\text{fatom-of-hatom } (p, ts) = (p, \text{fterms-of-hterms } ts)$

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

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

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

theorem $\text{fatom-of-hatom-hatom-of-fatom}$ [simp]: $\text{ground}_{ts} (\text{snd } l) \Longrightarrow \text{fatom-of-hatom } (\text{hatom-of-fatom } l) = l$ **by** ($\text{cases } l$) auto

lemma $\text{hatom-of-fatom-bij}$: $\text{bij-betw } \text{hatom-of-fatom } \text{ground-fatoms } \text{UNIV}$
unfolding bij-betw-def

proof

show $\text{inj-on } \text{hatom-of-fatom } \text{ground-fatoms}$ **using** $\text{inj-on-inverseI } \text{fatom-of-hatom-hatom-of-fatom}$
unfolding ground-fatoms-def

by ($\text{metis } (\text{mono-tags}, \text{lifting}) \text{mem-Collect-eq}$)

next

have $\forall a. \exists a'. \text{ground}_{ts} (\text{snd } a') \wedge a = \text{hatom-of-fatom } a'$

using $\text{ground-fatom-of-hatom } \text{hatom-of-fatom-fatom-of-hatom}$ **by** metis

then show $\text{hatom-of-fatom } ' \text{ground-fatoms} = \text{UNIV}$ **unfolding** ground-fatoms-def
by blast

qed

lemma $\text{fatom-of-hatom-bij}$: $\text{bij-betw } \text{fatom-of-hatom } \text{UNIV } \text{ground-fatoms}$
unfolding $\text{bij-betw-def } \text{inj-on-def}$

proof

show $\forall x \in \text{UNIV}. \forall y \in \text{UNIV}. \text{fatom-of-hatom } x = \text{fatom-of-hatom } y \longrightarrow x = y$

using $\text{ground-fatom-of-hatom } \text{hatom-of-fatom-fatom-of-hatom}$ **by** metis

next

have $\forall a. \text{ground}_{ts} (\text{snd } a) \longrightarrow (a = \text{fatom-of-hatom } (\text{hatom-of-fatom } a))$ **using**
 $\text{hatom-of-fatom-fatom-of-hatom}$ **by** auto

then have $\text{ground-fatoms} \subseteq \text{fatom-of-hatom } ' \text{UNIV}$ **unfolding** ground-fatoms-def
by blast

moreover

have $\forall l. \text{ground}_{ts} (\text{snd } (\text{fatom-of-hatom } l))$ **using** $\text{ground-fatom-of-hatom}$ **by**
 auto

ultimately show $\text{fatom-of-hatom } ' \text{UNIV} = \text{ground-fatoms}$

using $\text{hatom-of-fatom-fatom-of-hatom } \text{ground-fatom-of-hatom}$ **unfolding** ground-fatoms-def
by auto

qed

1.4 Enumerations

1.4.1 Enumerating Strings

definition $\text{nat-from-string} :: \text{string} \Rightarrow \text{nat}$ **where**

$\text{nat-from-string} \equiv (\text{SOME } f. \text{bij } f)$

definition $\text{string-from-nat}:: \text{nat} \Rightarrow \text{string}$ **where**
 $\text{string-from-nat} \equiv \text{inv nat-from-string}$

lemma $\text{nat-from-string-bij}: \text{bij nat-from-string}$

proof –

have $\text{countable} (\text{UNIV}::\text{string set})$ **by** *auto*

moreover

have $\text{infinite} (\text{UNIV}::\text{string set})$ **using** *infinite-UNIV-listI* **by** *auto*

ultimately

obtain x **where** $\text{bij} (x:: \text{string} \Rightarrow \text{nat})$ **using** $\text{countableE-infinite}[\text{of UNIV}]$ **by** *blast*

then show *?thesis* **unfolding** $\text{nat-from-string-def}$ **using** *someI* **by** *metis*

qed

lemma $\text{string-from-nat-bij}: \text{bij string-from-nat}$ **unfolding** $\text{string-from-nat-def}$ **using** $\text{nat-from-string-bij}$ bij-betw-inv-into **by** *auto*

lemma $\text{nat-from-string-string-from-nat}[\text{simp}]: \text{nat-from-string} (\text{string-from-nat } n)$
 $= n$

unfolding $\text{string-from-nat-def}$

using $\text{nat-from-string-bij}$ $f\text{-inv-into-f}[\text{of nat-from-string}]$ **by** *simp*

lemma $\text{string-from-nat-nat-from-string}[\text{simp}]: \text{string-from-nat} (\text{nat-from-string } n)$
 $= n$

unfolding $\text{string-from-nat-def}$

using $\text{nat-from-string-bij}$ $\text{inv-into-f-f}[\text{of nat-from-string}]$ **by** *simp*

1.4.2 Enumerating Herbrand Atoms

definition $\text{nat-from-hatom}:: \text{hterm atom} \Rightarrow \text{nat}$ **where**
 $\text{nat-from-hatom} \equiv (\text{SOME } f. \text{bij } f)$

definition $\text{hatom-from-nat}:: \text{nat} \Rightarrow \text{hterm atom}$ **where**
 $\text{hatom-from-nat} \equiv \text{inv nat-from-hatom}$

instantiation $\text{hterm} :: \text{countable}$ **begin**

instance **by** *countable-datatype*

end

lemma $\text{infinite-hatoms}: \text{infinite} (\text{UNIV} :: (\text{pred-sym} * 't \text{list}) \text{set})$

proof –

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

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

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

moreover

have $\forall n. ?diago n \in \text{UNIV}$ **by** *auto*

ultimately show *infinite* (*UNIV* :: (*pred-sym* * *t list*) *set*) **using** *infinity*[*of ?undiago ?diago UNIV*] **by** *simp*
qed

lemma *nat-from-hatom-bij*: *bij nat-from-hatom*

proof –

let *?S* = *UNIV* :: (*pred-sym* * (*t::countable*) *list*) *set*

have *countable ?S* **by** *auto*

moreover

have *infinite ?S* **using** *infinite-hatoms* **by** *auto*

ultimately

obtain *x* **where** *bij* (*x* :: *hterm atom* \Rightarrow *nat*) **using** *countableE-infinite*[*of ?S*]
by *blast*

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

then show *?thesis unfolding bij-betw-def inj-on-def unfolding nat-from-hatom-def by simp*

qed

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

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

unfolding *hatom-from-nat-def*

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

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

unfolding *hatom-from-nat-def*

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

1.4.3 Enumerating Ground Atoms

definition *fatom-from-nat* :: *nat* \Rightarrow *fterm atom* **where**

fatom-from-nat = ($\lambda n. \text{fatom-of-hatom (hatom-from-nat } n)$)

definition *nat-from-fatom* :: *fterm atom* \Rightarrow *nat* **where**

nat-from-fatom = ($\lambda t. \text{nat-from-hatom (hatom-of-fatom } t)$)

theorem *diag-undiag-fatom[simp]*: *ground_{t_s}* *ts* \Longrightarrow *fatom-from-nat (nat-from-fatom (p,ts)) = (p,ts)*

unfolding *fatom-from-nat-def nat-from-fatom-def by auto*

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

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

using *hatom-from-nat-bij bij-betw-trans fatom-of-hatom-bij hatom-from-nat-bij*

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

lemma *ground-fatome-from-nat*: $\text{ground}_{ts} (\text{snd} (\text{fatome-from-nat } x))$ **unfolding** *fatome-from-nat-def*
using *ground-fatome-of-hatome* **by** *auto*

lemma *nat-from-fatome-bij*: *bij-betw nat-from-fatome ground-fatoms UNIV*
using *nat-from-hatome-bij bij-betw-trans hatome-of-fatome-bij hatome-from-nat-bij*
unfolding *nat-from-fatome-def comp-def* **by** *blast*

end

2 Trees

theory *Tree* **imports** *Main* **begin**

Sometimes it is nice to think of *bools* as directions in a binary tree

hide-const (**open**) *Left Right*
type-synonym *dir* = *bool*
definition *Left* :: *bool* **where** *Left* = *True*
definition *Right* :: *bool* **where** *Right* = *False*
declare *Left-def* [*simp*]
declare *Right-def* [*simp*]

datatype *tree* =
 Leaf
| *Branching* (*ltree*: *tree*) (*rtree*: *tree*)

2.1 Sizes

fun *treesize* :: *tree* \Rightarrow *nat* **where**
 treesize Leaf = 0
| *treesize (Branching l r)* = 1 + *treesize l* + *treesize r*

lemma *treesize-Leaf*: $\text{treesize } T = 0 \implies T = \text{Leaf}$ **by** (*cases T*) *auto*

lemma *treesize-Branching*: $\text{treesize } T = \text{Suc } n \implies \exists l r. T = \text{Branching } l r$ **by**
(*cases T*) *auto*

2.2 Paths

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

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

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

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

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

lemma *path-inv-Branching*:

$\text{path } p (\text{Branching } l r) \longleftrightarrow (p = [] \vee (\exists a p'. p = a\#p' \wedge (a \longrightarrow \text{path } p' l) \wedge (\neg a \longrightarrow \text{path } p' r)))$ (**is** $?L \longleftrightarrow ?R$)

proof

assume $?L$ **then show** $?R$ **by** (*induction* p) *auto*

next

assume $r: ?R$

then show $?L$

proof

assume $p = []$ **then show** $?L$ **by** *auto*

next

assume $\exists a p'. p = a\#p' \wedge (a \longrightarrow \text{path } p' l) \wedge (\neg a \longrightarrow \text{path } p' r)$

then obtain $a p'$ **where** $p = a\#p' \wedge (a \longrightarrow \text{path } p' l) \wedge (\neg a \longrightarrow \text{path } p' r)$

by *auto*

then show $?L$ **by** (*cases* a) *auto*

qed

qed

lemma *path-prefix*: $\text{path } (ds1@ds2) T \Longrightarrow \text{path } ds1 T$

proof (*induction* $ds1$ *arbitrary: T*)

case (*Cons* $a ds1$)

then have $\exists l r. T = \text{Branching } l r$ **using** *path-inv-Leaf* **by** (*cases* T) *auto*

then obtain $l r$ **where** $p\text{-}lr: T = \text{Branching } l r$ **by** *auto*

show $?case$

proof (*cases* a)

assume $a\text{true}: a$

then have $\text{path } ((ds1) @ ds2) l$ **using** $p\text{-}lr$ *Cons(2)* *path-inv-Branching* **by**

auto

then have $\text{path } ds1 l$ **using** *Cons(1)* **by** *auto*

then show $\text{path } (a \# ds1) T$ **using** $p\text{-}lr$ $a\text{true}$ **by** *auto*

next

assume $a\text{false}: \neg a$

then have $\text{path } ((ds1) @ ds2) r$ **using** $p\text{-}lr$ *Cons(2)* *path-inv-Branching* **by**

auto

then have $\text{path } ds1 r$ **using** *Cons(1)* **by** *auto*

then show $\text{path } (a \# ds1) T$ **using** $p\text{-}lr$ $a\text{false}$ **by** *auto*

qed

next

case (Nil) then show ?case by auto
qed

2.3 Branches

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

lemma has-branch: $\exists b$. 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 \longleftrightarrow b = []

by (cases b) auto

lemma branch-inv-Branching-Left:

branch (Left#b) (Branching l r) \longleftrightarrow branch b l

by auto

lemma branch-inv-Branching-Right:

branch (Right#b) (Branching l r) \longleftrightarrow branch b r

by auto

lemma branch-inv-Branching:

branch b (Branching l r) \longleftrightarrow

($\exists a b'$. b=a#b' \wedge (a \longrightarrow branch b' l) \wedge (\neg a \longrightarrow branch b' r))

by (induction b) auto

lemma branch-inv-Leaf2:

T = Leaf \longleftrightarrow ($\forall b$. branch b T \longrightarrow b = [])

proof –

{

assume T=Leaf

then have $\forall b$. branch b T \longrightarrow b = [] **using** branch-inv-Leaf **by** auto

}

moreover

{

assume $\forall b$. branch b T \longrightarrow b = []

then have $\forall b$. branch b T \longrightarrow $\neg(\exists a b'$. b = a # b') **by** auto

then have $\forall b. \text{branch } b \ T \longrightarrow \neg(\exists l \ r. \text{branch } b \ (\text{Branching } l \ r))$
using *branch-inv-Branching* **by** *auto*
then have $T = \text{Leaf}$ **using** *has-branch[of T]* **by** (*metis branch.elims(2)*)
}
ultimately show $T = \text{Leaf} \longleftrightarrow (\forall b. \text{branch } b \ T \longrightarrow b = [])$ **by** *auto*
qed

lemma *branch-is-path*:

branch ds T \implies path ds T

proof (*induction T arbitrary: ds*)

case *Leaf*

then have $ds = []$ **using** *branch-inv-Leaf* **by** *auto*

then show *?case* **by** *auto*

next

case (*Branching T₁ T₂*)

then obtain $a \ b$ **where** $ds-p: ds = a \# b \wedge (a \longrightarrow \text{branch } b \ T_1) \wedge (\neg a \longrightarrow \text{branch } b \ T_2)$ **using** *branch-inv-Branching[of ds]* **by** *blast*

then have $(a \longrightarrow \text{path } b \ T_1) \wedge (\neg a \longrightarrow \text{path } b \ T_2)$ **using** *Branching* **by** *auto*

then show *?case* **using** *ds-p* **by** (*cases a*) *auto*

qed

lemma *Branching-Leaf-Leaf-Tree*: $T = \text{Branching } T_1 \ T_2 \implies (\exists B. \text{branch } (B @ [\text{True}]) \ T \wedge \text{branch } (B @ [\text{False}]) \ T)$

proof (*induction T arbitrary: T₁ T₂*)

case *Leaf* **then show** *?case* **by** *auto*

next

case (*Branching T₁' T₂'*)

{

assume $T_1' = \text{Leaf} \wedge T_2' = \text{Leaf}$

then have $\text{branch } ([] @ [\text{True}]) \ (\text{Branching } T_1' \ T_2') \wedge \text{branch } ([] @ [\text{False}]) \ (\text{Branching } T_1' \ T_2')$ **by** *auto*

then have *?case* **by** *metis*

}

moreover

{

fix $T_{11} \ T_{12}$

assume $T_1' = \text{Branching } T_{11} \ T_{12}$

then obtain B **where** $\text{branch } (B @ [\text{True}]) \ T_1'$

$\wedge \text{branch } (B @ [\text{False}]) \ T_1'$ **using** *Branching* **by** *blast*

then have $\text{branch } (([\text{True}] @ B) @ [\text{True}]) \ (\text{Branching } T_1' \ T_2')$

$\wedge \text{branch } (([\text{True}] @ B) @ [\text{False}]) \ (\text{Branching } T_1' \ T_2')$ **by** *auto*

then have *?case* **by** *blast*

}

moreover

{

fix $T_{11} \ T_{12}$

assume $T_2' = \text{Branching } T_{11} \ T_{12}$

then obtain B **where** $\text{branch } (B @ [\text{True}]) \ T_2'$

$\wedge \text{branch } (B @ [\text{False}]) \ T_2'$ **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*: \neg internal b Leaf **using** internal.simps **by** blast

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

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

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

proof

assume ?L **then show** ?R **by** (metis internal.simps(2) neq-Nil-conv)

next

assume r: ?R

then show ?L

proof

assume p = [] **then show** ?L **by** auto

next

assume ∃ a p'. p=a#p' ∧ (a ⟶ internal p' l) ∧ (¬a ⟶ internal p' r)

then obtain a p' **where** p=a#p' ∧ (a ⟶ internal p' l) ∧ (¬a ⟶ internal p' r) **by** auto

then show ?L **by** (cases a) auto

qed

qed

lemma *internal-is-path*:

 internal ds T ⟹ path ds T

proof (induction T arbitrary: ds)

case Leaf

then have False **using** internal-inv-Leaf **by** auto

then show ?case **by** auto

next

case (Branching T₁ T₂)

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

lemma *internal-prefix*: $\text{internal } (ds1 @ ds2 @ [d])\ T \Longrightarrow \text{internal } ds1\ T$

proof (*induction ds1 arbitrary: T*)

case (*Cons a ds1*)

then have $\exists l\ r. T = \text{Branching } l\ r$ **using** *internal-inv-Leaf* **by** (*cases T*) *auto*

then obtain $l\ r$ **where** $p\text{-}lr: T = \text{Branching } l\ r$ **by** *auto*

show *?case*

proof (*cases a*)

assume $atrue: a$

then have $\text{internal } ((ds1) @ ds2 @ [d])\ l$ **using** $p\text{-}lr$ *Cons(2)* *internal-inv-Branching*

by *auto*

then have $\text{internal } ds1\ l$ **using** *Cons(1)* **by** *auto*

then show $\text{internal } (a \# ds1)\ T$ **using** $p\text{-}lr$ $atrue$ **by** *auto*

next

assume $afalse: \sim a$

then have $\text{internal } ((ds1) @ ds2 @ [d])\ r$ **using** $p\text{-}lr$ *Cons(2)* *internal-inv-Branching*

by *auto*

then have $\text{internal } ds1\ r$ **using** *Cons(1)* **by** *auto*

then show $\text{internal } (a \# ds1)\ T$ **using** $p\text{-}lr$ $afalse$ **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** *auto*

qed

lemma *internal-branch*: $\text{branch } (ds1 @ ds2 @ [d])\ T \Longrightarrow \text{internal } ds1\ T$

proof (*induction ds1 arbitrary: T*)

case (*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** $p\text{-}lr: T = \text{Branching } l\ r$ **by** *auto*

show *?case*

proof (*cases a*)

assume $atrue: a$

then have $\text{branch } (ds1 @ ds2 @ [d])\ l$ **using** $p\text{-}lr$ *Cons(2)* *branch-inv-Branching*

by *auto*

then have $\text{internal } ds1\ l$ **using** *Cons(1)* **by** *auto*

then show $\text{internal } (a \# ds1)\ T$ **using** $p\text{-}lr$ $atrue$ **by** *auto*

next

assume $afalse: \sim a$

then have $\text{branch } ((ds1) @ ds2 @ [d])\ r$ **using** $p\text{-}lr$ *Cons(2)* *branch-inv-Branching*

by *auto*

```

    then have internal ds1 r using Cons(1) by auto
    then show internal (a # ds1) T using p-lr afalse by auto
  qed
next
case (Nil)
then have  $\exists l r. T = \text{Branching } l r$  using branch-inv-Leaf by (cases T) auto
then show ?case by auto
qed

```

```

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

```

2.5 Deleting Nodes

```

fun delete :: dir list  $\Rightarrow$  tree  $\Rightarrow$  tree where
  delete [] T = Leaf
| delete (True#ds) (Branching T1 T2) = Branching (delete ds T1) T2
| delete (False#ds) (Branching T1 T2) = Branching T1 (delete ds T2)
| delete (a#ds) Leaf = Leaf

```

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

lemma path-delete: path p (delete ds T) \implies path p T

proof (induction p arbitrary: T ds)

case Nil

then show ?case by simp

next

case (Cons a p)

then obtain b ds' where bds'-p: ds=b#ds' by (cases ds) auto

have $\exists dT1 dT2. \text{delete } ds T = \text{Branching } dT1 dT2$ using Cons path-inv-Cons by auto

then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto

then have $\exists T1 T2. T = \text{Branching } T1 T2$

by (cases T; cases ds) auto

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

```

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

```

```

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

```

lemma *branch-delete*: $\text{branch } p \text{ (delete ds } T) \implies \text{branch } p \text{ } T \vee p=ds$

proof (*induction p arbitrary: T ds*)

case *Nil*

then have $\text{delete ds } T = \text{Leaf}$ by (*cases delete ds T*) *auto*

then have $ds = [] \vee T = \text{Leaf}$ using *delete.elims* by *blast*

then show ?case by *auto*

next

case (*Cons a p*)

then obtain $b \text{ } ds'$ where $bds'-p: ds=b\#ds'$ by (*cases ds*) *auto*

have $\exists dT1 \text{ } dT2. \text{delete ds } T = \text{Branching } dT1 \text{ } dT2$ using *Cons path-inv-Cons branch-is-path* by *blast*

then obtain $dT1 \text{ } dT2$ where $\text{delete ds } T = \text{Branching } dT1 \text{ } dT2$ by *auto*

then have $\exists T1 \text{ } T2. T = \text{Branching } T1 \text{ } T2$


```

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

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

```

lemma branch-delete-postfix: path p (delete ds T) \implies $\neg(\exists c cs. p = ds @ c\#cs)$

```

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

  have  $\exists dT1 dT2. \text{delete } ds \ T = \text{Branching } dT1 \ dT2$  using Cons path-inv-Cons
  by auto
  then obtain dT1 dT2 where delete ds T = Branching dT1 dT2 by auto

  then have  $\exists T1 \ T2. T = \text{Branching } T1 \ T2$ 
    by (cases T; cases ds) auto
  then obtain T1 T2 where T1T2-p: T=Branching T1 T2 by auto

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

```

```

lemma treezise-delete: internal p T  $\implies$  treezise (delete p T) < treezise T
proof (induction p arbitrary: T)
  case (Nil)
    then have  $\exists T1 T2. T = \text{Branching } T1 T2$  by (cases T) auto
    then obtain T1 T2 where T1T2-p:  $T = \text{Branching } T1 T2$  by auto
    then show ?case by auto
  next
    case (Cons a p)
    then have  $\exists T1 T2. T = \text{Branching } T1 T2$  using path-inv-Cons internal-is-path
by blast
    then obtain T1 T2 where T1T2-p:  $T = \text{Branching } T1 T2$  by auto
    show ?case
      proof (cases a)
        assume a-p: a
        from a-p have  $\text{delete } (a\#p) T = (\text{Branching } (\text{delete } p T1) T2)$  using
T1T2-p by auto
        moreover
        from a-p have internal p T1 using T1T2-p Cons by auto
        then have  $\text{treezise } (\text{delete } p T1) < \text{treezise } T1$  using Cons by auto
        ultimately
        show ?thesis using T1T2-p by auto
      next
        assume a-p:  $\neg a$ 
        from a-p have  $\text{delete } (a\#p) T = (\text{Branching } T1 (\text{delete } p T2))$  using T1T2-p
by auto
        moreover
        from a-p have internal p T2 using T1T2-p Cons by auto
        then have  $\text{treezise } (\text{delete } p T2) < \text{treezise } T2$  using Cons by auto
        ultimately
        show ?thesis using T1T2-p by auto
    qed
qed

```

```

fun cutoff :: (dir list  $\Rightarrow$  bool)  $\Rightarrow$  dir list  $\Rightarrow$  tree  $\Rightarrow$  tree where
  cutoff red ds (Branching T1 T2) =
    (if red ds then Leaf else Branching (cutoff red (ds@[Left]) T1) (cutoff red
    (ds@[Right]) T2))
  | cutoff red ds Leaf = Leaf

```

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

```

lemma treezise-cutoff: treezise (cutoff red ds T)  $\leq$  treezise T
proof (induction T arbitrary: ds)
  case Leaf then show ?case by auto
next
  case (Branching T1 T2)

```

then have $\text{treesize } (\text{cutoff red } (ds@[Left]) T1) + \text{treesize } (\text{cutoff red } (ds@[Right]) T2) \leq \text{treesize } T1 + \text{treesize } T2$ **using** *add-mono* **by** *blast*
then show *?case* **by** *auto*
qed

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

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

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

lemma *cutoff-branch'*:

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

proof (*induction T arbitrary: ds*)

case (*Leaf*)

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

{

fix b

assume $\text{branch } b ?T$

then have $\text{branch } b \text{ Leaf}$ **by** *auto*

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

}

then show *?case* **by** *simp*

next

case (*Branching* $T_1 T_2$)

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

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

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

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

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

using *Branching* **by** *blast*

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

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

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

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

using *Branching* **by** *blast*

{

fix b

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

have $\text{red } ds \vee \neg \text{red } ds$ **by** *auto*

then have $\text{red}(ds@b)$

```

proof
  assume  $ds-p$ :  $red\ ds$ 
  then have  $?T = Leaf$  by auto
  then have  $b = []$  using  $b-p$  branch-inv-Leaf by auto
  then show  $red(ds@b)$  using  $ds-p$  by auto
next
  assume  $ds-p$ :  $\neg red\ ds$ 
  let  $?T_1' = cutoff\ red\ (ds@[Left])\ T_1$ 
  let  $?T_2' = cutoff\ red\ (ds@[Right])\ T_2$ 
  from  $ds-p$  have  $?T = Branching\ ?T_1'\ ?T_2'$  by auto
  from this  $b-p$  obtain  $a\ b'$  where  $b = a \# b' \wedge (a \longrightarrow branch\ b'\ ?T_1') \wedge$ 
 $(\neg a \longrightarrow branch\ b'\ ?T_2')$  using branch-inv-Branching[of  $b\ ?T_1'\ ?T_2'$ ] by auto
  then show  $red(ds@b)$  using  $aa\ bb$  by (cases  $a$ ) auto
  qed
}
then show  $?case$  by blast
qed

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

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

  from Branching have  $\forall p. branch\ (Right\#p)\ (Branching\ T_1\ T_2) \longrightarrow red\ (ds\ @$ 
 $(Right\#p))$  by blast
  then have  $\forall p. branch\ p\ T_2 \longrightarrow red\ (ds\ @\ (Right\#p))$  by auto
  then have  $anybranch\ T_2\ (\lambda p. red\ ((ds\ @\ [Right])\ @\ p))$  by auto
  then have  $bb$ :  $anyinternal\ (cutoff\ red\ (ds\ @\ [Right])\ T_2)\ (\lambda p. \neg red\ ((ds\ @$ 
 $[Right])\ @\ p))$  using Branching by blast
  {
    fix  $p$ 
    assume  $b-p$ : internal  $p\ ?T$ 
    then have  $ds-p$ :  $\neg red\ ds$  using internal-inv-Leaf by auto
    have  $p=[] \vee p\neq[]$  by auto
    then have  $\neg red(ds@p)$ 
    proof

```

```

    assume p=[] then show ¬red(ds@p) using ds-p by auto
next
let ?T1' = cutoff red (ds@[Left]) T1
let ?T2' = cutoff red (ds@[Right]) T2
assume p≠[]
moreover
have ?T = Branching ?T1' ?T2' using ds-p by auto
ultimately
obtain a p' where b-p: p = a # p' ∧
  (a → internal p' (cutoff red (ds @ [Left]) T1)) ∧
  (¬ a → internal p' (cutoff red (ds @ [Right]) T2))
  using b-p internal-inv-Branching[of p ?T1' ?T2'] by auto
then have ¬red(ds @ [a] @ p') using aa bb by (cases a) auto
then show ¬red(ds @ p) using b-p by simp
qed
}
then show ?case by blast
qed

```

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

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

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

3 Possibly Infinite Trees

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

abbreviation *wf-tree* :: *dir list set* \Rightarrow *bool* **where**
wf-tree T \equiv ($\forall ds d. (ds @ d) \in T \longrightarrow ds \in T$)

The subtree in with root r

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

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

lemma *subtree-pos*:
subtree T ds \subseteq *subtree* T (ds @ [Left]) \cup *subtree* T (ds @ [Right]) \cup {ds}

```

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

```

3.1 Infinite Paths

```

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

```

```

lemma infpath-length: wf-infpath f ⇒ length (f n) = n

```

```

proof (induction n)
  case 0 then show ?case by auto
next
  case (Suc n) then show ?case by (metis length-append-singleton)
qed

```

```

lemma chain-prefix: wf-infpath f ⇒ n1 ≤ n2 ⇒ ∃ a. (f n1) @ a = (f n2)

```

```

proof (induction n2)
  case (Suc n2)
  then have n1 ≤ n2 ∨ n1 = Suc n2 by auto
  then show ?case
  proof
    assume n1 ≤ n2
    then obtain a where a: f n1 @ a = f n2 using Suc by auto
    have b: ∃ b. f (Suc n2) = f n2 @ [b] using Suc by auto
    from a b have ∃ b. f n1 @ (a @ [b]) = f (Suc n2) by auto
    then show ∃ c. f n1 @ c = f (Suc n2) by blast
  next
    assume n1 = Suc n2
    then have f n1 @ [] = f (Suc n2) by auto
    then show ∃ a. f n1 @ a = f (Suc n2) by auto
  qed
qed auto

```

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

```

lemma ith-in-extension:
  assumes chain: wf-infpath f
  assumes smalli: i < length (f n1)
  assumes n1n2: n1 ≤ n2

```

shows $f\ n_1\ !\ i = f\ n_2\ !\ i$
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** $a\text{-}p: f\ n_1\ @\ a = f\ n_2$ **by** *auto*
have $(f\ n_1\ @\ a)\ !\ i = f\ n_1\ !\ i$ **using** *smalli* **by** (*simp add: nth-append*)
then show *?thesis* **using** $a\text{-}p$ **by** *auto*
qed

4 König's Lemma

lemma *inf-subst*:

assumes *inf*: $\neg\text{finite}(\text{subtree } T\ ds)$

shows $\neg\text{finite}(\text{subtree } T\ (ds\ @\ [Left])) \vee \neg\text{finite}(\text{subtree } T\ (ds\ @\ [Right]))$

proof –

let $?subtree = \text{subtree } T$

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

then show $\neg\text{finite} (?subtree\ (ds\ @\ [Left])) \vee \neg\text{finite} (?subtree\ (ds\ @\ [Right]))$

using *inf* **by** *auto*

qed

fun *buildchain* :: $(dir\ list \Rightarrow dir\ list) \Rightarrow nat \Rightarrow dir\ list$ **where**

buildchain *next* 0 = []

| *buildchain* *next* (Suc n) = *next* (*buildchain* *next* n)

lemma *konig*:

assumes *inf*: $\neg\text{finite } T$

assumes *wellformed*: *wf-tree* T

shows $\exists c. \text{wf-infpth } c \wedge (\forall n. (c\ n) \in T)$

proof

let $?subtree = \text{subtree } T$

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

let $?c = \text{buildchain } ?nextnode$

have *is-chain*: *wf-infpth* $?c$ **by** *auto*

from *wellformed* **have** *prefix*: $\bigwedge ds\ d. (ds\ @\ d) \in T \implies ds \in T$ **by** *blast*

{
fix n
have $(?c\ n) \in T \wedge \neg\text{finite} (?subtree\ (?c\ n))$
proof (*induction* n)


```

    case 0
    have  $\exists ds. ds \in T$  using inf by (simp add: not-finite-existsD)
    then obtain ds where  $ds \in T$  by auto
    then have  $([]@ds) \in T$  by auto
    then have  $[] \in T$  using prefix[of  $[]$ ] by auto
    then show ?case using inf by auto
next
case (Suc n)
from Suc have next-in:  $(?c\ n) \in T$  by auto
from Suc have next-inf:  $\neg$ finite (?subtree (?c n)) by auto

from next-inf have next-next-inf:
   $\neg$ finite (?subtree (?nextnode (?c n)))
  using inf-subs by auto
then have  $\exists ds. ds \in ?subtree\ (?nextnode\ (?c\ n))$ 
  by (simp add: not-finite-existsD)
then obtain ds where dss:  $ds \in ?subtree\ (?nextnode\ (?c\ n))$  by auto
then have  $ds \in T \exists$  suf.  $ds = (?nextnode\ (?c\ n)) @$  suf by auto
then obtain suf where  $ds \in T \wedge ds = (?nextnode\ (?c\ n)) @$  suf by auto
then have  $(?nextnode\ (?c\ n)) \in T$ 
  using prefix[of ?nextnode (?c n) suf] by auto

then have  $(?c\ (Suc\ n)) \in T$  by auto
then show ?case using next-next-inf by auto
qed
}
then show wf-infpath ?c  $\wedge (\forall n. (?c\ n) \in T)$  using is-chain by auto
qed
end

```

5 More Terms and Literals

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

fun *complement* :: *'t literal* \Rightarrow *'t literal* (*-^c* [300] 300) **where**

(*Pos P ts*)^{*c*} = *Neg P ts*
| (*Neg P ts*)^{*c*} = *Pos P ts*

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

lemma *cancel-comp2*:

assumes *asm*: $l_1^c = l_2^c$

shows $l_1 = l_2$

proof –

from *asm* **have** $(l_1^c)^c = (l_2^c)^c$ **by** *auto*

then have $l_1 = (l_2^c)^c$ **using** *cancel-comp1*[of l_1] **by** *auto*

then show *?thesis* **using** *cancel-comp1*[of l_2] **by** *auto*

qed

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

lemma *comp-exi2*: $\exists l. l' = l^c$

proof

show $l' = (l^c)^c$ **using** *cancel-comp1*[*of l'*] **by** *auto*
qed

lemma *comp-swap*: $l_1^c = l_2 \longleftrightarrow l_1 = l_2^c$

proof –

have $l_1^c = l_2 \implies l_1 = l_2^c$ **using** *cancel-comp1*[*of l₁*] **by** *auto*
moreover
have $l_1 = l_2^c \implies l_1^c = l_2$ **using** *cancel-comp1* **by** *auto*
ultimately
show *?thesis* **by** *auto*
qed

lemma *sign-comp*: $sign\ l_1 \neq sign\ l_2 \wedge get\text{-pred}\ l_1 = get\text{-pred}\ l_2 \wedge get\text{-terms}\ l_1 = get\text{-terms}\ l_2 \longleftrightarrow l_2 = l_1^c$
by (*cases l₁*; *cases l₂*) *auto*

lemma *sign-comp-atom*: $sign\ l_1 \neq sign\ l_2 \wedge get\text{-atom}\ l_1 = get\text{-atom}\ l_2 \longleftrightarrow l_2 = l_1^c$
by (*cases l₁*; *cases l₂*) *auto*

6 Clauses

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

abbreviation *complementls* :: *'t literal set* \Rightarrow *'t literal set* ($-^C$ [300] 300) **where**
 $L^C \equiv complement\ 'L$

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

lemma *cancel-compls2*:

assumes *asm*: $L_1^C = L_2^C$

shows $L_1 = L_2$

proof –

from *asm* **have** $(L_1^C)^C = (L_2^C)^C$ **by** *auto*

then show *?thesis* **using** *cancel-compls1*[*of L₁*] *cancel-compls1*[*of L₂*] **by** *simp*
qed

fun *vars_t* :: *fterm* \Rightarrow *var-sym set* **where**

$vars_t\ (Var\ x) = \{x\}$

$| vars_t\ (Fun\ f\ ts) = (\bigcup t \in set\ ts.\ vars_t\ t)$

abbreviation $vars_{ts} :: fterm\ list \Rightarrow var\text{-}sym\ set$ **where**
 $vars_{ts}\ ts \equiv (\bigcup t \in set\ ts.\ vars_t\ t)$

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

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

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

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

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

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

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

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

7 Semantics

type-synonym $'u\ fun\text{-}denot = fun\text{-}sym \Rightarrow 'u\ list \Rightarrow 'u$

type-synonym $'u\ pred\text{-}denot = pred\text{-}sym \Rightarrow 'u\ list \Rightarrow bool$

type-synonym $'u\ var\text{-}denot = var\text{-}sym \Rightarrow 'u$

fun $eval_t :: 'u\ var\text{-}denot \Rightarrow 'u\ fun\text{-}denot \Rightarrow fterm \Rightarrow 'u$ **where**

$eval_t\ E\ F\ (Var\ x) = E\ x$

| $eval_t\ E\ F\ (Fun\ f\ ts) = F\ f\ (map\ (eval_t\ E\ F)\ ts)$

abbreviation $eval_{ts} :: 'u\ var\text{-}denot \Rightarrow 'u\ fun\text{-}denot \Rightarrow fterm\ list \Rightarrow 'u\ list$ **where**

$eval_{ts}\ E\ F\ ts \equiv map\ (eval_t\ E\ F)\ ts$

fun $eval_l :: 'u\ var\text{-}denot \Rightarrow 'u\ fun\text{-}denot \Rightarrow 'u\ pred\text{-}denot \Rightarrow fterm\ literal \Rightarrow bool$
where

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

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

definition $eval_c :: 'u\ fun\text{-}denot \Rightarrow 'u\ pred\text{-}denot \Rightarrow fterm\ clause \Rightarrow bool$ **where**

$eval_c\ F\ G\ C \longleftrightarrow (\forall E.\ \exists l \in C.\ eval_l\ E\ F\ G\ l)$

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

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

7.1 Semantics of Ground Terms

lemma *ground-var-denott*: $ground_t t \implies (eval_t E F t = eval_t E' F t)$

proof (*induction t*)

case (*Var x*)

then have *False by auto*

then show *?case by auto*

next

case (*Fun f ts*)

then have $\forall t \in set\ ts. ground_t t$ **by auto**

then have $\forall t \in set\ ts. eval_t E F t = eval_t E' F t$ **using Fun by auto**

then have $eval_{ts} E F ts = eval_{ts} E' F ts$ **by auto**

then have $F f (map (eval_t E F) ts) = F f (map (eval_t E' F) ts)$ **by metis**

then show *?case by simp*

qed

lemma *ground-var-denotts*: $ground_{ts} ts \implies (eval_{ts} E F ts = eval_{ts} E' F ts)$

using *ground-var-denott by (metis map-eq-conv)*

lemma *ground-var-denot*: $ground_l l \implies (eval_l E F G l = eval_l E' F G l)$

proof (*induction l*)

case *Pos* **then show** *?case using ground-var-denotts by (metis eval_l.simps(1) literal.sel(3))*

next

case *Neg* **then show** *?case using ground-var-denotts by (metis eval_l.simps(2) literal.sel(4))*

qed

8 Substitutions

type-synonym *substitution* = *var-sym* \Rightarrow *fterm*

fun *sub* :: *fterm* \Rightarrow *substitution* \Rightarrow *fterm* (**infixl** \cdot_t 55) **where**

 (*Var x*) $\cdot_t \sigma = \sigma x$

| (*Fun f ts*) $\cdot_t \sigma = Fun f (map (\lambda t. t \cdot_t \sigma) ts)$

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

$ts \cdot_{ts} \sigma \equiv (map (\lambda t. t \cdot_t \sigma) ts)$

fun *subl* :: *fterm literal* \Rightarrow *substitution* \Rightarrow *fterm literal* (**infixl** \cdot_l 55) **where**

 (*Pos p ts*) $\cdot_l \sigma = Pos p (ts \cdot_{ts} \sigma)$

| (*Neg p ts*) $\cdot_l \sigma = Neg p (ts \cdot_{ts} \sigma)$

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

$L \cdot_{ls} \sigma \equiv (\lambda l. l \cdot_l \sigma) \text{ ` } L$

lemma *subls-def2*: $L \cdot_{ls} \sigma = \{l \cdot_l \sigma \mid l. l \in L\}$ **by** *auto*

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

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

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

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

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

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

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

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

8.1 The Empty Substitution

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

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

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

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

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

lemma *instance-of_t-self*: $instance-of_t t t$
unfolding *instance-of_t-def*
proof

show $t = t \cdot_t \varepsilon$ **using** *empty-subt* **by** *auto*
qed

lemma *instance-of_{ts}-self*: *instance-of_{ts} ts ts*
unfolding *instance-of_{ts}-def*

proof

show $ts = ts \cdot_{ts} \varepsilon$ **using** *empty-subts* **by** *auto*
qed

lemma *instance-of_l-self*: *instance-of_l l l*
unfolding *instance-of_l-def*

proof

show $l = l \cdot_l \varepsilon$ **using** *empty-subl* **by** *auto*
qed

lemma *instance-of_{ls}-self*: *instance-of_{ls} L L*
unfolding *instance-of_{ls}-def*

proof

show $L = L \cdot_{ls} \varepsilon$ **using** *empty-subls* **by** *auto*
qed

8.2 Substitutions and Ground Terms

lemma *ground-sub*: *ground_t t \implies t \cdot_t $\sigma = t$*
by (*induction t*) (*auto simp add: map-idI*)

lemma *ground-sub_s*: *ground_{ts} ts \implies ts \cdot_{ts} $\sigma = ts$*
using *ground-sub* **by** (*simp add: map-idI*)

lemma *ground_l-sub_s*: *ground_l l \implies l \cdot_l $\sigma = l$*
using *ground-sub_s* **by** (*cases l*) *auto*

lemma *ground_{ls}-sub_s*:

assumes *ground*: *ground_{ls} L*

shows $L \cdot_{ls} \sigma = L$

proof –

{
 fix l
 assume $l-L$: $l \in L$
 then have *ground_l l* **using** *ground* **by** *auto*
 then have $l = l \cdot_l \sigma$ **using** *ground_l-sub_s* **by** *auto*
 moreover
 then have $l \cdot_l \sigma \in L \cdot_{ls} \sigma$ **using** $l-L$ **by** *auto*
 ultimately
 have $l \in L \cdot_{ls} \sigma$ **by** *auto*

}

moreover

{

fix l

```

    assume l-L: l ∈ L ·ls σ
    then obtain l'-p: l' ∈ L ∧ l' ·l σ = l by auto
    then have l' = l using ground groundl-subs by auto
    from l-L l'-p this have l ∈ L by auto
  }
  ultimately show ?thesis by auto
qed

```

8.3 Composition

definition *composition* :: *substitution* ⇒ *substitution* ⇒ *substitution* (**infixl** · 55)
where

$$(\sigma_1 \cdot \sigma_2) x = (\sigma_1 x) \cdot_t \sigma_2$$

lemma *composition-conseq2t*: $(t \cdot_t \sigma_1) \cdot_t \sigma_2 = t \cdot_t (\sigma_1 \cdot \sigma_2)$

proof (*induction t*)

case (*Var x*)

have $((\text{Var } x) \cdot_t \sigma_1) \cdot_t \sigma_2 = (\sigma_1 x) \cdot_t \sigma_2$ **by** *simp*

also have $\dots = (\sigma_1 \cdot \sigma_2) x$ **unfolding** *composition-def* **by** *simp*

finally show ?*case* **by** *auto*

next

case (*Fun t ts*)

then show ?*case* **unfolding** *composition-def* **by** *auto*

qed

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

using *composition-conseq2t* **by** *auto*

lemma *composition-conseq2l*: $(l \cdot_l \sigma_1) \cdot_l \sigma_2 = l \cdot_l (\sigma_1 \cdot \sigma_2)$

using *composition-conseq2t* **by** (*cases l*) *auto*

lemma *composition-conseq2ls*: $(L \cdot_{ls} \sigma_1) \cdot_{ls} \sigma_2 = L \cdot_{ls} (\sigma_1 \cdot \sigma_2)$

using *composition-conseq2l* **apply** *auto*

apply (*metis imageI*)

done

lemma *composition-assoc*: $\sigma_1 \cdot (\sigma_2 \cdot \sigma_3) = (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$

proof

fix *x*

show $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3)) x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3) x$

by (*simp only: composition-def composition-conseq2t*)

qed

lemma *empty-comp1*: $(\sigma \cdot \varepsilon) = \sigma$

proof

fix *x*

show $(\sigma \cdot \varepsilon) x = \sigma x$ **unfolding** *composition-def* **using** *empty-subst* **by** *auto*

qed

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

lemma *instance-of_t-trans* :
 assumes t_{12} : *instance-of_t* t_1 t_2
 assumes t_{23} : *instance-of_t* t_2 t_3
 shows *instance-of_t* t_1 t_3
proof –
 from t_{12} **obtain** σ_{12} **where** $t_1 = t_2 \cdot_t \sigma_{12}$
 unfolding *instance-of_t-def* **by** *auto*
 moreover
 from t_{23} **obtain** σ_{23} **where** $t_2 = t_3 \cdot_t \sigma_{23}$
 unfolding *instance-of_t-def* **by** *auto*
 ultimately
 have $t_1 = (t_3 \cdot_t \sigma_{23}) \cdot_t \sigma_{12}$ **by** *auto*
 then have $t_1 = t_3 \cdot_t (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2t* **by** *simp*
 then show *thesis* **unfolding** *instance-of_t-def* **by** *auto*
qed

lemma *instance-of_{ts}-trans* :
 assumes ts_{12} : *instance-of_{ts}* ts_1 ts_2
 assumes ts_{23} : *instance-of_{ts}* ts_2 ts_3
 shows *instance-of_{ts}* ts_1 ts_3
proof –
 from ts_{12} **obtain** σ_{12} **where** $ts_1 = ts_2 \cdot_{ts} \sigma_{12}$
 unfolding *instance-of_{ts}-def* **by** *auto*
 moreover
 from ts_{23} **obtain** σ_{23} **where** $ts_2 = ts_3 \cdot_{ts} \sigma_{23}$
 unfolding *instance-of_{ts}-def* **by** *auto*
 ultimately
 have $ts_1 = (ts_3 \cdot_{ts} \sigma_{23}) \cdot_{ts} \sigma_{12}$ **by** *auto*
 then have $ts_1 = ts_3 \cdot_{ts} (\sigma_{23} \cdot \sigma_{12})$ **using** *composition-conseq2ts* **by** *simp*
 then show *thesis* **unfolding** *instance-of_{ts}-def* **by** *auto*
qed

lemma *instance-of_l-trans* :
 assumes l_{12} : *instance-of_l* l_1 l_2
 assumes l_{23} : *instance-of_l* l_2 l_3
 shows *instance-of_l* l_1 l_3
proof –
 from l_{12} **obtain** σ_{12} **where** $l_1 = l_2 \cdot_l \sigma_{12}$
 unfolding *instance-of_l-def* **by** *auto*
 moreover
 from l_{23} **obtain** σ_{23} **where** $l_2 = l_3 \cdot_l \sigma_{23}$
 unfolding *instance-of_l-def* **by** *auto*

ultimately
 have $l_1 = (l_3 \cdot_l \sigma_{23}) \cdot_l \sigma_{12}$ by *auto*
 then have $l_1 = l_3 \cdot_l (\sigma_{23} \cdot \sigma_{12})$ using *composition-conseq2l* by *simp*
 then show *?thesis unfolding instance-of_l-def* by *auto*
 qed

lemma *instance-of_l_s-trans* :
 assumes L_{12} : *instance-of_l_s* L_1 L_2
 assumes L_{23} : *instance-of_l_s* L_2 L_3
 shows *instance-of_l_s* L_1 L_3
 proof –
 from L_{12} obtain σ_{12} where $L_1 = L_2 \cdot_{l_s} \sigma_{12}$
 unfolding *instance-of_l_s-def* by *auto*
 moreover
 from L_{23} obtain σ_{23} where $L_2 = L_3 \cdot_{l_s} \sigma_{23}$
 unfolding *instance-of_l_s-def* by *auto*
 ultimately
 have $L_1 = (L_3 \cdot_{l_s} \sigma_{23}) \cdot_{l_s} \sigma_{12}$ by *auto*
 then have $L_1 = L_3 \cdot_{l_s} (\sigma_{23} \cdot \sigma_{12})$ using *composition-conseq2ls* by *simp*
 then show *?thesis unfolding instance-of_l_s-def* by *auto*
 qed

8.4 Merging substitutions

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

lemma *relevant-vars-subt*:

$\forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x \implies t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
proof (*induction t*)
case (*Fun f ts*)
have $f: \bigwedge t. t \in \text{set } ts \implies \text{vars}_t t \subseteq \text{vars}_{ts} ts$ **by** (*induction ts*) *auto*
have $\forall t \in \text{set } ts. t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
proof
fix t
assume $tints: t \in \text{set } ts$
then have $\forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x$ **using** f *Fun*(\mathcal{Q}) **by** *auto*
then show $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$ **using** *Fun tints* **by** *auto*
qed
then have $ts \cdot_{ts} \sigma_1 = ts \cdot_{ts} \sigma_2$ **by** *auto*
then show *?case* **by** *auto*
qed *auto*

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

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

lemma *relevant-vars-subls*:
assumes $asm: \forall x \in \text{vars}_{ls} L. \sigma_1 x = \sigma_2 x$
shows $L \cdot_{ls} \sigma_1 = L \cdot_{ls} \sigma_2$
proof –
have $f: \bigwedge l. l \in L \implies \text{vars}_l l \subseteq \text{vars}_{ls} L$ **unfolding** *vars_ls-def* **by** *auto*
have $\forall l \in L. l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
proof
fix l
assume $linls: l \in L$

then have $\forall x \in \text{vars}_l \ l. \sigma_1 x = \sigma_2 x$ **using** *f asm* **by** *auto*
then show $l \cdot_l \sigma_1 = l \cdot_l \sigma_2$ **using** *relevant-vars-subl linls* **by** *auto*
qed
then show *?thesis* **by** (*meson image-cong*)
qed

lemma *merge-sub*:

assumes *dist*: $\text{vars}_{l_s} C \cap \text{vars}_{l_s} D = \{\}$
assumes *CC'*: $C \cdot_{l_s} \text{lmbd} = C'$
assumes *DD'*: $D \cdot_{l_s} \mu = D'$
shows $\exists \eta. C \cdot_{l_s} \eta = C' \wedge D \cdot_{l_s} \eta = D'$
proof –
let $?\eta = \lambda x. \text{if } x \in \text{vars}_{l_s} C \text{ then } \text{lmbd } x \text{ else } \mu x$
have $\forall x \in \text{vars}_{l_s} C. ?\eta x = \text{lmbd } x$ **by** *auto*
then have $C \cdot_{l_s} ?\eta = C \cdot_{l_s} \text{lmbd}$ **using** *relevant-vars-subls*[of $C \ ?\eta \ \text{lmbd}$] **by** *auto*
then have $C \cdot_{l_s} ?\eta = C'$ **using** *CC'* **by** *auto*
moreover
have $\forall x \in \text{vars}_{l_s} D. ?\eta x = \mu x$ **using** *dist* **by** *auto*
then have $D \cdot_{l_s} ?\eta = D \cdot_{l_s} \mu$ **using** *relevant-vars-subls*[of $D \ ?\eta \ \mu$] **by** *auto*
then have $D \cdot_{l_s} ?\eta = D'$ **using** *DD'* **by** *auto*
ultimately
show *?thesis* **by** *auto*
qed

8.5 Standardizing apart

abbreviation *std₁* :: *fterm clause* \Rightarrow *fterm clause* **where**
 $\text{std}_1 C \equiv C \cdot_{l_s} (\lambda x. \text{Var } ("1" @ x))$

abbreviation *std₂* :: *fterm clause* \Rightarrow *fterm clause* **where**
 $\text{std}_2 C \equiv C \cdot_{l_s} (\lambda x. \text{Var } ("2" @ x))$

lemma *std-apart-apart''*:

$x \in \text{vars}_t \ (t \cdot_t (\lambda x. \text{char list. Var } (y @ x))) \Longrightarrow \exists x'. x = y @ x'$
by (*induction t*) *auto*

lemma *std-apart-apart'*: $x \in \text{vars}_l \ (l \cdot_l (\lambda x. \text{Var } (y @ x))) \Longrightarrow \exists x'. x = y @ x'$
unfolding *vars_l-def* **using** *std-apart-apart''* **by** (*cases l*) *auto*

lemma *std-apart-apart*: $\text{vars}_{l_s} (\text{std}_1 C_1) \cap \text{vars}_{l_s} (\text{std}_2 C_2) = \{\}$

proof –

$\{$
fix x
assume *xin*: $x \in \text{vars}_{l_s} (\text{std}_1 C_1) \cap \text{vars}_{l_s} (\text{std}_2 C_2)$
from *xin* **have** $x \in \text{vars}_{l_s} (\text{std}_1 C_1)$ **by** *auto*
then have $\exists x'. x = "1" @ x'$
using *std-apart-apart'*[of $x \ - \ "1"$] **unfolding** *vars_{l_s}-def* **by** *auto*

moreover
from xin **have** $x \in vars_{l_s} (std_2 C_2)$ **by** *auto*
then have $\exists x'. x = ''2'' @x'$
 using *std-apart-apart* [of $x - ''2''$] **unfolding** *vars_{l_s}*-def **by** *auto*
 ultimately have *False* **by** *auto*
 then have $x \in \{\}$ **by** *auto*
 }
then show *?thesis* **by** *auto*
qed

lemma *std-apart-instance-of_{l_s}*1: instance-of_{l_s C₁ (std₁ C₁)}

proof –

have *empty*: $(\lambda x. Var (''1''@x)) \cdot (\lambda x. Var (tl x)) = \varepsilon$ **using** *composition-def*
by *auto*

have $C_1 \cdot_{l_s} \varepsilon = C_1$ **using** *empty-subls* **by** *auto*

then have $C_1 \cdot_{l_s} ((\lambda x. Var (''1''@x)) \cdot (\lambda x. Var (tl x))) = C_1$ **using** *empty* **by**
auto

then have $(C_1 \cdot_{l_s} (\lambda x. Var (''1''@x))) \cdot_{l_s} (\lambda x. Var (tl x)) = C_1$ **using** *composition-conseq2ls*
by *auto*

then have $C_1 = (std_1 C_1) \cdot_{l_s} (\lambda x. Var (tl x))$ **by** *auto*

then show *instance-of_{l_s}* C₁ (std₁ C₁) **unfolding** *instance-of_{l_s}*-def **by** *auto*

qed

lemma *std-apart-instance-of_{l_s}*2: instance-of_{l_s C₂ (std₂ C₂)}

proof –

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

have $C_2 \cdot_{l_s} \varepsilon = C_2$ **using** *empty-subls* **by** *auto*

then have $C_2 \cdot_{l_s} ((\lambda x. Var (''2''@x)) \cdot (\lambda x. Var (tl x))) = C_2$ **using** *empty*
by *auto*

then have $(C_2 \cdot_{l_s} (\lambda x. Var (''2''@x))) \cdot_{l_s} (\lambda x. Var (tl x)) = C_2$ **using** *composition-conseq2ls*
by *auto*

then have $C_2 = (std_2 C_2) \cdot_{l_s} (\lambda x. Var (tl x))$ **by** *auto*

then show *instance-of_{l_s}* C₂ (std₂ C₂) **unfolding** *instance-of_{l_s}*-def **by** *auto*

qed

9 Unifiers

definition *unifier_{ts}* :: *substitution* \Rightarrow *fterm set* \Rightarrow *bool* **where**

unifier_{ts} σ $ts \longleftrightarrow (\exists t'. \forall t \in ts. t \cdot_t \sigma = t')$

definition *unifier_{l_s}* :: *substitution* \Rightarrow *fterm literal set* \Rightarrow *bool* **where**

unifier_{l_s} σ $L \longleftrightarrow (\exists l'. \forall l \in L. l \cdot_{l_s} \sigma = l')$

lemma *unif-sub*:

assumes *unif*: *unifier_{l_s}* σ 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 $L \cdot_{l_s} \sigma = \{l \cdot_l \sigma\}$ **unfolding** *unifier_{l_s}*-def by *auto*
 then show *?thesis* by *auto*
qed

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

lemma *unifier_{l_s}*-def2:
 assumes *L-lem*: $L \neq \{\}$
 shows *unifier_{l_s}* σ $L \longleftrightarrow (\exists l. L \cdot_{l_s} \sigma = \{l\})$
proof
 assume *unif*: *unifier_{l_s}* σ L
 from *L-lem* obtain l where $l \in L$ by *auto*
 then have $L \cdot_{l_s} \sigma = \{l \cdot_l \sigma\}$ **using** *unif* **unfolding** *unifier_{l_s}*-def by *auto*
 then show $\exists l. L \cdot_{l_s} \sigma = \{l\}$ by *auto*
next
 assume $\exists l. L \cdot_{l_s} \sigma = \{l\}$
 then obtain l where $L \cdot_{l_s} \sigma = \{l\}$ by *auto*
 then have $\forall l' \in L. l' \cdot_{l'} \sigma = l$ by *auto*
 then show *unifier_{l_s}* σ L **unfolding** *unifier_{l_s}*-def by *auto*
qed

lemma *ground_{l_s}*-unif-singleton:
 assumes *ground_{l_s}*: *ground_{l_s}* L
 assumes *unif*: *unifier_{l_s}* $\sigma' L$
 assumes *empt*: $L \neq \{\}$
 shows $\exists l. L = \{l\}$
proof –
 from *unif empt* have $\exists l. L \cdot_{l_s} \sigma' = \{l\}$ **using** *unif-sub* by *auto*
 then show *?thesis* **using** *ground_{l_s}*-subls *ground_{l_s}* by *auto*
qed

definition *unifiablets* :: *fterm set* \Rightarrow *bool* where

$unifiablets fs \longleftrightarrow (\exists \sigma. unifier_{ts} \sigma fs)$

definition $unifiablets :: fterm literal set \Rightarrow bool$ **where**
 $unifiablets L \longleftrightarrow (\exists \sigma. unifier_{ts} \sigma L)$

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

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

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

next

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

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

qed

lemma $unifier-sub1: unifier_{ts} \sigma L \Longrightarrow L' \subseteq L \Longrightarrow unifier_{ts} \sigma L'$
unfolding $unifier_{ts}-def$ **by** $auto$

lemma $unifier-sub2:$

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

proof –

have $L_1 \subseteq (L_1 \cup L_2) \wedge L_2 \subseteq (L_1 \cup L_2)$ **by** $simp$
from $this$ asm **show** $?thesis$ **using** $unifier-sub1$ **by** $auto$

qed

9.1 Most General Unifiers

definition $mgu_{ts} :: substitution \Rightarrow fterm set \Rightarrow bool$ **where**

$$mgu_{ts} \sigma ts \longleftrightarrow \text{unifier}_{ts} \sigma ts \wedge (\forall u. \text{unifier}_{ts} u ts \longrightarrow (\exists i. u = \sigma \cdot i))$$

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

10 Resolution

definition $\text{applicable} :: \text{fterm clause} \Rightarrow \text{fterm clause}$
 $\Rightarrow \text{fterm literal set} \Rightarrow \text{fterm literal set}$
 $\Rightarrow \text{substitution} \Rightarrow \text{bool}$ **where**

$$\begin{aligned} \text{applicable } C_1 C_2 L_1 L_2 \sigma &\longleftrightarrow \\ C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\} \\ \wedge \text{vars}_{ls} C_1 \cap \text{vars}_{ls} C_2 &= \{\} \\ \wedge L_1 \subseteq C_1 \wedge L_2 \subseteq C_2 \\ \wedge mgu_{ls} \sigma (L_1 \cup L_2^C) \end{aligned}$$

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

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

inductive $\text{mresolution-step} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool}$ **where**
 mresolution-rule:

$$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow \text{applicable } C_1 C_2 L_1 L_2 \sigma \Longrightarrow \\ \text{mresolution-step } Cs (Cs \cup \{\text{mresolution } C_1 C_2 L_1 L_2 \sigma\})$$

| $\text{standardize-apart:}$

$$C \in Cs \Longrightarrow \text{var-renaming-of } C C' \Longrightarrow \text{mresolution-step } Cs (Cs \cup \{C'\})$$

inductive $\text{resolution-step} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool}$ **where**
 resolution-rule:

$$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow \text{applicable } C_1 C_2 L_1 L_2 \sigma \Longrightarrow \\ \text{resolution-step } Cs (Cs \cup \{\text{resolution } C_1 C_2 L_1 L_2 \sigma\})$$

| $\text{standardize-apart:}$

$$C \in Cs \Longrightarrow \text{var-renaming-of } C C' \Longrightarrow \text{resolution-step } Cs (Cs \cup \{C'\})$$

definition $\text{mresolution-deriv} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool}$ **where**
 $\text{mresolution-deriv} = \text{rtranclp mresolution-step}$

definition $\text{resolution-deriv} :: \text{fterm clause set} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool}$ **where**
 $\text{resolution-deriv} = \text{rtranclp resolution-step}$

11 Soundness

definition $evalsub :: 'u \text{ var-denot} \Rightarrow 'u \text{ fun-denot} \Rightarrow \text{substitution} \Rightarrow 'u \text{ var-denot}$
where

$$evalsub \ E \ F \ \sigma = eval_t \ E \ F \ \circ \ \sigma$$

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

apply (*induction t*)

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

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

done

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

using *substitutiont* **by** *auto*

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

apply (*induction l*)

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

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

done

lemma *subst-sound*:

assumes *asm*: $eval_c \ F \ G \ C$

shows $eval_c \ F \ G \ (C \cdot_{ls} \ \sigma)$

proof –

have $\forall E. \exists l \in C \cdot_{ls} \ \sigma. eval_l \ E \ F \ G \ l$

proof

fix E

from *asm* **have** $\forall E. \exists l \in C. eval_l \ E \ F \ G \ l$ **unfolding** *eval_c-def* **by** *auto*

then **have** $\exists l \in C. eval_l \ (evalsub \ E \ F \ \sigma) \ F \ G \ l$ **by** *auto*

then **show** $\exists l \in C \cdot_{ls} \ \sigma. eval_l \ E \ F \ G \ l$ **using** *substitution* **by** *blast*

qed

then **show** $eval_c \ F \ G \ (C \cdot_{ls} \ \sigma)$ **unfolding** *eval_c-def* **by** *auto*

qed

lemma *simple-resolution-sound*:

assumes $C_1 \text{sat}$: $eval_c \ F \ G \ C_1$

assumes $C_2 \text{sat}$: $eval_c \ F \ G \ C_2$

assumes $l_1 \text{inc}_1$: $l_1 \in C_1$

assumes $l_2 \text{inc}_2$: $l_2 \in C_2$

assumes *comp*: $l_1^c = l_2$

shows $eval_c \ F \ G \ ((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))$

proof –

have $\forall E. \exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))). eval_l \ E \ F \ G \ l$

proof

fix E

have $eval_l \ E \ F \ G \ l_1 \vee eval_l \ E \ F \ G \ l_2$ **using** *comp* **by** (*cases l₁*) *auto*

then **show** $\exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))). eval_l \ E \ F \ G \ l$

proof

assume $eval_l E F G l_1$
then have $\neg eval_l E F G l_2$ **using** *comp* **by** (*cases* l_1) *auto*
then have $\exists l_2' \in C_2. l_2' \neq l_2 \wedge eval_l E F G l_2'$ **using** $l_2 inc_2 C_2 sat$
unfolding *eval_c-def* **by** *auto*
then show $\exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}). eval_l E F G l$ **by** *auto*
next
assume $eval_l E F G l_2$
then have $\neg eval_l E F G l_1$ **using** *comp* **by** (*cases* l_1) *auto*
then have $\exists l_1' \in C_1. l_1' \neq l_1 \wedge eval_l E F G l_1'$ **using** $l_1 inc_1 C_1 sat$
unfolding *eval_c-def* **by** *auto*
then show $\exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}). eval_l E F G l$ **by** *auto*
qed
qed
then show *?thesis* **unfolding** *eval_c-def* **by** *simp*
qed

lemma *mresolution-sound*:

assumes $sat_1: eval_c F G C_1$
assumes $sat_2: eval_c F G C_2$
assumes *appl*: *applicable* $C_1 C_2 L_1 L_2 \sigma$
shows $eval_c F G (mresolution C_1 C_2 L_1 L_2 \sigma)$

proof –

from sat_1 **have** $sat_1 \sigma: eval_c F G (C_1 \cdot_{l_s} \sigma)$ **using** *subst-sound* **by** *blast*
from sat_2 **have** $sat_2 \sigma: eval_c F G (C_2 \cdot_{l_s} \sigma)$ **using** *subst-sound* **by** *blast*

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

from l_1-p *appl* **have** $l_1 \in C_1$ **unfolding** *applicable-def* **by** *auto*
then have $inc_1 \sigma: l_1 \cdot_l \sigma \in C_1 \cdot_{l_s} \sigma$ **by** *auto*

from l_1-p **have** $unified_1: l_1 \in (L_1 \cup (L_2^C))$ **by** *auto*

from l_1-p *appl* **have** $l_1 \sigma is l_1 \sigma: \{l_1 \cdot_l \sigma\} = L_1 \cdot_{l_s} \sigma$
unfolding *mgu_{l_s}-def* *unifier_{l_s}-def* *applicable-def* **by** *auto*

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

from l_2-p *appl* **have** $l_2 \in C_2$ **unfolding** *applicable-def* **by** *auto*
then have $inc_2 \sigma: l_2 \cdot_l \sigma \in C_2 \cdot_{l_s} \sigma$ **by** *auto*

from l_2-p **have** $unified_2: l_2^c \in (L_1 \cup (L_2^C))$ **by** *auto*

from $unified_1$ $unified_2$ *appl* **have** $l_1 \cdot_l \sigma = (l_2^c) \cdot_l \sigma$
unfolding *mgu_{l_s}-def* *unifier_{l_s}-def* *applicable-def* **by** *auto*
then have *comp*: $(l_1 \cdot_l \sigma)^c = l_2 \cdot_l \sigma$ **using** *comp-sub* *comp-swap* **by** *auto*

from *appl* **have** *unifier_{l_s}* $\sigma (L_2^C)$
using *unifier-sub2* **unfolding** *mgu_{l_s}-def* *applicable-def* **by** *blast*
then have *unifier_{l_s}* σL_2 **by** *auto*

from *this* l_2 - p **have** $l_2\sigma isl_2\sigma: \{l_2 \cdot_l \sigma\} = L_2 \cdot_{l_s} \sigma$ **unfolding** *unifier_{l_s}-def* **by** *auto*

from $sat_1\sigma$ $sat_2\sigma$ $inc_1\sigma$ $inc_2\sigma$ *comp* **have** $eval_c F G ((C_1 \cdot_{l_s} \sigma) - \{l_1 \cdot_l \sigma\} \cup ((C_2 \cdot_{l_s} \sigma) - \{l_2 \cdot_l \sigma\}))$ **using** *simple-resolution-sound*[*of* $F G C_1 \cdot_{l_s} \sigma C_2 \cdot_{l_s} \sigma l_1 \cdot_l \sigma l_2 \cdot_l \sigma$] **by** *auto*

from *this* $l_1\sigma isl_1\sigma$ $l_2\sigma isl_2\sigma$ **show** *?thesis* **unfolding** *mresolution-def* **by** *auto* **qed**

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

lemma *superset-sound*:

assumes *sup*: $C \subseteq C'$

assumes *sat*: $eval_c F G C$

shows $eval_c F G C'$

proof –

have $\forall E. \exists l \in C'. eval_l E F G l$

proof

fix E

from *sat* **have** $\forall E. \exists l \in C. eval_l E F G l$ **unfolding** *eval_c-def* **by** –

then **have** $\exists l \in C. eval_l E F G l$ **by** *auto*

then **show** $\exists l \in C'. eval_l E F G l$ **using** *sup* **by** *auto*

qed

then **show** $eval_c F G C'$ **unfolding** *eval_c-def* **by** *auto*

qed

lemma *resolution-sound*:

assumes *sat₁*: $eval_c F G C_1$

assumes *sat₂*: $eval_c F G C_2$

assumes *appl*: *applicable* $C_1 C_2 L_1 L_2 \sigma$

shows $eval_c F G (resolution C_1 C_2 L_1 L_2 \sigma)$

proof –

from *sat₁* *sat₂* *appl* **have** $eval_c F G (mresolution C_1 C_2 L_1 L_2 \sigma)$ **using** *mresolution-sound* **by** *blast*

then **show** *?thesis* **using** *superset-sound* *resolution-superset* **by** *metis*

qed

lemma *sound-step*: $mresolution\text{-}step Cs Cs' \implies eval_{c_s} F G Cs \implies eval_{c_s} F G Cs'$

proof (*induction rule*: *mresolution-step.induct*)

case (*mresolution-rule* $C_1 Cs C_2 l_1 l_2 \sigma$)

then **have** $eval_c F G C_1 \wedge eval_c F G C_2$ **unfolding** *eval_{c_s}-def* **by** *auto*

then **have** $eval_c F G (mresolution C_1 C_2 l_1 l_2 \sigma)$

using *mresolution-sound* *mresolution-rule* **by** *auto*

then **show** *?case* **using** *mresolution-rule* **unfolding** *eval_{c_s}-def* **by** *auto*

next
case (*standardize-apart* C Cs C')
then have $eval_c F G C$ **unfolding** *eval_{cs}-def* **by** *auto*
then have $eval_c F G C'$ **using** *subst-sound standardize-apart unfolding var-renaming-of-def instance-of_{1s}-def* **by** *metis*
then show ?*case* **using** *standardize-apart unfolding eval_{cs}-def* **by** *auto*
qed

lemma *lsound-step: resolution-step* $Cs Cs' \implies eval_{cs} F G Cs \implies eval_{cs} F G Cs'$

proof (*induction rule: resolution-step.induct*)

case (*resolution-rule* $C_1 Cs C_2 l_1 l_2 \sigma$)

then have $eval_c F G C_1 \wedge eval_c F G C_2$ **unfolding** *eval_{cs}-def* **by** *auto*

then have $eval_c F G$ (*resolution* $C_1 C_2 l_1 l_2 \sigma$)

using *resolution-sound resolution-rule* **by** *auto*

then show ?*case* **using** *resolution-rule unfolding eval_{cs}-def* **by** *auto*

next

case (*standardize-apart* $C Cs C'$)

then have $eval_c F G C$ **unfolding** *eval_{cs}-def* **by** *auto*

then have $eval_c F G C'$ **using** *subst-sound standardize-apart unfolding var-renaming-of-def instance-of_{1s}-def* **by** *metis*

then show ?*case* **using** *standardize-apart unfolding eval_{cs}-def* **by** *auto*

qed

lemma *sound-derivation:*

mresolution-deriv $Cs Cs' \implies eval_{cs} F G Cs \implies eval_{cs} F G Cs'$

unfolding *mresolution-deriv-def*

proof (*induction rule: rtranclp.induct*)

case *rtrancl-refl* **then show** ?*case* **by** *auto*

next

case (*rtrancl-into-rtrancl* $Cs_1 Cs_2 Cs_3$) **then show** ?*case* **using** *sound-step* **by** *auto*

qed

lemma *lsound-derivation:*

resolution-deriv $Cs Cs' \implies eval_{cs} F G Cs \implies eval_{cs} F G Cs'$

unfolding *resolution-deriv-def*

proof (*induction rule: rtranclp.induct*)

case *rtrancl-refl* **then show** ?*case* **by** *auto*

next

case (*rtrancl-into-rtrancl* $Cs_1 Cs_2 Cs_3$) **then show** ?*case* **using** *lsound-step* **by** *auto*

qed

12 Herbrand Interpretations

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

term $HFun$

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

lemma *eval-ground_{ts}*: $ground_{ts} ts \implies (eval_{ts} E \text{ HFun } ts) = \text{hterms-of-fters } ts$
unfolding *hterms-of-fters-def* **using** *eval-ground_t* **by** (*induction ts*) *auto*

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

13 Partial Interpretations

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

definition *falsifies_l* :: *partial-pred-denot* \Rightarrow *fterm literal* \Rightarrow *bool* **where**
 $falsifies_l G l \longleftrightarrow$
 $ground_l l$
 $\wedge (let i = nat-from-fatome (get-atom l) in$
 $i < length G \wedge G ! i = (\neg sign l)$
 $)$

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

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

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

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

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

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

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

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

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

lemma *ground-sub-of-denotl*: $\text{ground}_l (l \cdot_l \text{sub-of-denot } E)$

proof –

have $\text{ground}_{ts} (\text{subs } (\text{get-terms } l) (\text{sub-of-denot } E))$

using *ground-sub-of-denotts* by auto

then show *?thesis* by (cases l) auto

qed

lemma *sub-of-denot-equivx*: $\text{eval}_t E \text{HFun } (\text{sub-of-denot } E x) = E x$

proof –

have $\text{ground}_t (\text{sub-of-denot } E x)$ using *ground-fterm-of-hterm* by simp

then

have $\text{eval}_t E \text{HFun } (\text{sub-of-denot } E x) = \text{hterm-of-fterm } (\text{sub-of-denot } E x)$

using *eval-ground_t(1)* by auto

also have $\dots = \text{hterm-of-fterm } (\text{fterm-of-hterm } (E x))$ by auto

also have $\dots = E x$ by auto

finally show *?thesis* by auto

qed

lemma *sub-of-denot-equivt*:

$\text{eval}_t E \text{HFun } (t \cdot_t (\text{sub-of-denot } E)) = \text{eval}_t E \text{HFun } t$

using *sub-of-denot-equivx* by (induction t) auto

lemma *sub-of-denot-equivts*: $\text{eval}_{ts} E \text{HFun } (ts \cdot_{ts} (\text{sub-of-denot } E)) = \text{eval}_{ts} E \text{HFun } ts$

using *sub-of-denot-equivt* by simp

lemma *sub-of-denot-equivl*: $\text{eval}_l E \text{HFun } G (l \cdot_l \text{sub-of-denot } E) \longleftrightarrow \text{eval}_l E \text{HFun } G l$

proof (induction l)

case (Pos p ts)

have $\text{eval}_l E \text{HFun } G ((\text{Pos } p \text{ ts}) \cdot_l \text{sub-of-denot } E) \longleftrightarrow G p (\text{eval}_{ts} E \text{HFun } (ts \cdot_{ts} (\text{sub-of-denot } E)))$ by auto

also have $\dots \longleftrightarrow G p (\text{eval}_{ts} E \text{HFun } ts)$ using *sub-of-denot-equivts[of E ts]*
by metis

also have $\dots \longleftrightarrow \text{eval}_l E \text{HFun } G (\text{Pos } p \text{ ts})$ by simp

finally

show *?case* by blast

next

case (Neg p ts)

have $\text{eval}_l E \text{HFun } G ((\text{Neg } p \text{ ts}) \cdot_l \text{sub-of-denot } E) \longleftrightarrow \neg G p (\text{eval}_{ts} E \text{HFun } (ts \cdot_{ts} (\text{sub-of-denot } E)))$ by auto

also have $\dots \longleftrightarrow \neg G p (\text{eval}_{ts} E \text{HFun } ts)$ using *sub-of-denot-equivts[of E ts]*

by *metis*
also have $\dots = \text{eval}_l E \text{ HFun } G (Neg p ts)$ **by** *simp*
finally
show *?case* **by** *blast*
qed

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

lemma *sub-of-denot-equiv-ground'*:
 $\text{eval}_l E \text{ HFun } G l = \text{eval}_l E \text{ HFun } G (l \cdot_l \text{sub-of-denot } E) \wedge \text{ground}_l (l \cdot_l \text{sub-of-denot } E)$
using *sub-of-denot-equivl ground-sub-of-denotl* **by** *auto*

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

lemma *partial-equiv-subst*:
assumes $\text{falsifies}_c G (C \cdot_{ls} \tau)$
shows $\text{falsifies}_c G C$
proof –
from *assms* **obtain** C' **where** $C'-p$: $\text{instance-of}_{ls} C' (C \cdot_{ls} \tau) \wedge \text{falsifies}_g G C'$ **by** *auto*
then have $\text{instance-of}_{ls} (C \cdot_{ls} \tau) C$ **unfolding** *instance-of_{ls}-def* **by** *auto*
then have $\text{instance-of}_{ls} C' C$ **using** $C'-p$ *instance-of_{ls}-trans* **by** *auto*
then show *?thesis* **using** $C'-p$ **by** *auto*
qed

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

lemma *sub-of-denot-equiv-ground*:
 $((\exists l \in C. \text{eval}_l E \text{ HFun } G l) \longleftrightarrow (\exists l \in C \cdot_{ls} \text{sub-of-denot } E. \text{eval}_l E \text{ HFun } G l))$
 $\wedge \text{ground}_{ls} (C \cdot_{ls} \text{sub-of-denot } E)$
using *sub-of-denot-equiv-ground'* **by** *auto*

lemma *std₁-falsifies*: $\text{falsifies}_c G C_1 \longleftrightarrow \text{falsifies}_c G (\text{std}_1 C_1)$
proof
assume *asm*: $\text{falsifies}_c G C_1$
then obtain Cg **where** $\text{instance-of}_{ls} Cg C_1 \wedge \text{falsifies}_g G Cg$ **by** *auto*
moreover
then have $\text{instance-of}_{ls} Cg (\text{std}_1 C_1)$ **using** *std-apart-instance-of_{ls}1* *instance-of_{ls}-trans* *asm* **by** *blast*
ultimately
show $\text{falsifies}_c G (\text{std}_1 C_1)$ **by** *auto*
next
assume *asm*: $\text{falsifies}_c G (\text{std}_1 C_1)$
then have *inst*: $\text{instance-of}_{ls} (\text{std}_1 C_1) C_1$ **unfolding** *instance-of_{ls}-def* **by** *auto*

from *asm* **obtain** *Cg* **where** *instance-of_{1s} Cg (std₁ C₁) ∧ falsifies_g G Cg* **by**
auto
moreover
then have *instance-of_{1s} Cg C₁* **using** *inst instance-of_{1s}-trans* **by** *blast*
ultimately
show *falsifies_c G C₁* **by** *auto*
qed

lemma *std₂-falsifies*: *falsifies_c G C₂ ↔ falsifies_c G (std₂ C₂)*

proof

assume *asm: falsifies_c G C₂*
then obtain *Cg* **where** *instance-of_{1s} Cg C₂ ∧ falsifies_g G Cg* **by** *auto*
moreover
then have *instance-of_{1s} Cg (std₂ C₂)* **using** *std-apart-instance-of_{1s}2 instance-of_{1s}-trans*
asm **by** *blast*
ultimately
show *falsifies_c G (std₂ C₂)* **by** *auto*
next
assume *asm: falsifies_c G (std₂ C₂)*
then have *inst: instance-of_{1s} (std₂ C₂) C₂* **unfolding** *instance-of_{1s}-def* **by**
auto

from *asm* **obtain** *Cg* **where** *instance-of_{1s} Cg (std₂ C₂) ∧ falsifies_g G Cg* **by**
auto
moreover
then have *instance-of_{1s} Cg C₂* **using** *inst instance-of_{1s}-trans* **by** *blast*
ultimately
show *falsifies_c G C₂* **by** *auto*
qed

lemma *std₁-renames*: *var-renaming-of C₁ (std₁ C₁)*

proof –

have *instance-of_{1s} C₁ (std₁ C₁)* **using** *std-apart-instance-of_{1s}1* **by** *auto*
moreover have *instance-of_{1s} (std₁ C₁) C₁* **unfolding** *instance-of_{1s}-def* **by**
auto
ultimately show *var-renaming-of C₁ (std₁ C₁)* **unfolding** *var-renaming-of-def*
by *auto*
qed

lemma *std₂-renames*: *var-renaming-of C₂ (std₂ C₂)*

proof –

have *instance-of_{1s} C₂ (std₂ C₂)* **using** *std-apart-instance-of_{1s}2* **by** *auto*
moreover have *instance-of_{1s} (std₂ C₂) C₂* **unfolding** *instance-of_{1s}-def* **by**
auto
ultimately show *var-renaming-of C₂ (std₂ C₂)* **unfolding** *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 $G T Cs \equiv \text{branch } G T \wedge \text{falsifies}_{cs} G Cs$

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

open-branch $G T Cs \equiv \text{branch } G T \wedge \neg \text{falsifies}_{cs} G Cs$

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

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

15 Herbrand's Theorem

lemma *maximum*:

assumes *asm*: *finite C*

shows $\exists n :: \text{nat}. \forall l \in C. fl \leq n$

proof

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

qed

lemma *extend-preserves-model*:

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

assumes *C-ground*: *ground_{l_s} C*

assumes *C-sat*: $\neg \text{falsifies}_c (f (\text{Suc } n)) C$

assumes *n-max*: $\forall l \in C. \text{nat-from-fatomb} (\text{get-atom } l) \leq n$

shows *eval_c HFun* (*extend f*) *C*

proof –

let *?F* = *HFun*

let *?G* = *extend f*

{

fix *E*

from *C-sat* **have** $\forall C'. (\neg \text{instance-of}_{l_s} C' C \vee \neg \text{falsifies}_g (f (\text{Suc } n)) C')$ **by**

auto

then have $\neg \text{falsifies}_g (f (\text{Suc } n)) C$ **using** *instance-of_{l_s}-self* **by** *auto*

then obtain *l* **where** *l-p*: $l \in C \wedge \neg \text{falsifies}_l (f (\text{Suc } n)) l$ **using** *C-ground* **by**

blast

let *?i* = *nat-from-fatomb* (*get-atom l*)

from *l-p* **have** *i-n*: $?i \leq n$ **using** *n-max* **by** *auto*

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

auto

have *eval_l E HFun* (*extend f*) *l*

proof (*cases l*)

case (*Pos P ts*)

from *Pos l-p C-ground* **have** *ts-ground*: *ground_{t_s} ts* **by** *auto*


```

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

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

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

    then show ?thesis using Neg evall-groundts[of ts - ?G P] ts-ground by
auto
  qed
  then have  $\exists l \in C. eval_l E HFun (extend f) l$  using l-p by auto
}
  then have evalc HFun (extend f) C unfolding evalc-def by auto
  then show ?thesis using instance-ofls-self by auto
qed

lemma extend-preserves-model2:
  assumes f-infpath: wf-infpath (f :: nat  $\Rightarrow$  partial-pred-denot)
  assumes C-ground: groundls C
  assumes fin-c: finite C
  assumes model-C:  $\forall n. \neg$ falsifiesc (f n) C
  shows C-false: evalc HFun (extend f) C
proof -
  — Since C is finite, C has a largest index of a literal.
  obtain n where largest:  $\forall l \in C. nat-from-fatom (get-atom l) \leq n$  using fin-c
  maximum[of C  $\lambda l. nat-from-fatom (get-atom l)$ ] by blast
  moreover
  then have  $\neg$ falsifiesc (f (Suc n)) C using model-C by auto
  ultimately show ?thesis using model-C f-infpath C-ground extend-preserves-model[of
f C n ] by blast
qed

```

lemma *extend-infpth*:

assumes *f-infpth*: $wf\text{-infpth } (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes *model-c*: $\forall n. \neg falsifies_c (f n) C$
assumes *fin-c*: *finite C*
shows $eval_c \text{ HFun } (extend f) C$
unfolding *eval_c-def* **proof**
fix *E*
let *?G* = *extend f*
let *?σ* = *sub-of-denot E*

from *fin-c* **have** *fin-cσ*: *finite (C ·_{1s} sub-of-denot E)* **by** *auto*
have *groundcσ*: $ground_{1s} (C \cdot_{1s} sub\text{-of-denot } E)$ **using** *sub-of-denot-equiv-ground*
by *auto*

— Here starts the proof
— We go from syntactic FO world to syntactic ground world:
from *model-c* **have** $\forall n. \neg falsifies_c (f n) (C \cdot_{1s} ?\sigma)$ **using** *partial-equiv-subst* **by**
blast
— Then from syntactic ground world to semantic ground world:
then have $eval_c \text{ HFun } ?G (C \cdot_{1s} ?\sigma)$ **using** *groundcσ f-infpth fin-cσ extend-preserves-model2*[of
f C ·_{1s} ?σ] **by** *blast*
— Then from semantic ground world to semantic FO world:
then have $\forall E. \exists l \in (C \cdot_{1s} ?\sigma). eval_l E \text{ HFun } ?G l$ **unfolding** *eval_c-def* **by**
auto
then have $\exists l \in (C \cdot_{1s} ?\sigma). eval_l E \text{ HFun } ?G l$ **by** *auto*
then show $\exists l \in C. eval_l E \text{ HFun } ?G l$ **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 *infpth-model*:

assumes *f-infpth*: $wf\text{-infpth } (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes *model-cs*: $\forall n. \neg falsifies_{cs} (f n) Cs$
assumes *fin-cs*: *finite Cs*
assumes *fin-c*: $\forall C \in Cs. finite C$
shows $eval_{cs} \text{ HFun } (extend f) Cs$
proof —
let *?F* = *HFun*

have $\forall C \in Cs. eval_c ?F (extend f) C$
proof (*rule ballI*)
fix *C*
assume *asm*: $C \in Cs$
then have $\forall n. \neg falsifies_c (f n) C$ **using** *model-cs* **by** *auto*
then show $eval_c ?F (extend f) C$ **using** *fin-c asm f-infpth extend-infpth*[of
f C] **by** *auto*
qed
then show $eval_{cs} ?F (extend f) Cs$ **unfolding** *eval_{cs}-def* **by** *auto*

qed

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

lemma *branch-length*: *branch* b (*deeptree* n) \implies *length* b = n

proof (*induction* n *arbitrary*: b)

case 0 **then show** ?*case* **using** *branch-inv-Leaf* **by** *auto*

next

case (*Suc* n)

then have *branch* b (*Branching* (*deeptree* n) (*deeptree* n)) **by** *auto*

then obtain a b' **where** p: b = a#b' \wedge *branch* b' (*deeptree* n) **using** *branch-inv-Branching*[of b] **by** *blast*

then have *length* b' = n **using** *Suc* **by** *auto*

then show ?*case* **using** p **by** *auto*

qed

lemma *infinity*:

assumes *inj*: \forall n :: *nat*. *undiago* (*diago* n) = n

assumes *all-tree*: \forall n :: *nat*. (*diago* n) \in *tree*

shows \neg *finite tree*

proof –

from *inj all-tree* **have** \forall n. n = *undiago* (*diago* n) \wedge (*diago* n) \in *tree* **by** *auto*

then have \forall n. \exists ds. n = *undiago* ds \wedge ds \in *tree* **by** *auto*

then have *undiago* ' *tree* = (*UNIV* :: *nat set*) **by** *auto*

then have \neg *finite tree* **by** (*metis finite-imageI infinite-UNIV-nat*)

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

qed

lemma *longer-falsifies_l*:

assumes *falsifies_l* ds l

shows *falsifies_l* (ds@d) l

proof –

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

from *assms* **have** *i-p*: *ground_l* l \wedge ?i < *length* ds \wedge ds ! ?i = (\neg *sign* l) **unfolding** *falsifies_l-def* **by** *meson*

moreover

from *i-p* **have** ?i < *length* (ds@d) **by** *auto*

moreover

from *i-p* **have** (ds@d) ! ?i = (\neg *sign* l) **by** (*simp add: nth-append*)

ultimately

show ?*thesis* **unfolding** *falsifies_l-def* **by** *simp*

qed

lemma *longer-falsifies_g*:

assumes *falsifies_g* ds C

shows *falsifies_g* (ds @ d) C

proof –

```

{
  fix l
  assume  $l \in C$ 
  then have  $\text{falsifies}_l (ds @ d) l$  using assms longer-falsifiesl by auto
} then show ?thesis using assms by auto
qed

```

```

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

```

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

```

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

```

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

```

theorem herbrand':
  assumes openb:  $\forall T. \exists G. \text{open-branch } G T Cs$ 
  assumes finite-cs:  $\text{finite } Cs \vee C \in Cs. \text{finite } C$ 
  shows  $\exists G. \text{eval}_{c_s} \text{HFun } G Cs$ 
proof -
  — Show T infinite:
  let ?tree =  $\{G. \neg \text{falsifies}_{c_s} G Cs\}$ 
  let ?undia = length
  let ?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:  $\forall l. \text{open-branch } (?diag\ l) (\text{deeptree } l) Cs$ 
    using someI-ex[of  $\lambda b. \text{open-branch } b (\text{deeptree } -) Cs$ ] by auto
  then have  $\forall n. ?undia (?diag\ n) = n$  using branch-length by auto
  moreover
  have  $\forall n. (?diag\ n) \in ?tree$  using diag-open by auto
  ultimately
  have  $\neg \text{finite } ?tree$  using infinity[of  $-\lambda n. \text{SOME } b. \text{open-branch } b (-\ n) Cs$ ] by
simp

```

— Get infinite path:
moreover
have $\forall ds\ d. \neg \text{falsifies}_{cs} (ds\ @\ d)\ Cs \longrightarrow \neg \text{falsifies}_{cs}\ ds\ Cs$
using *longer-falsifies*[of *Cs*] **by** *blast*
then have $(\forall ds\ d. ds\ @\ d \in ?tree \longrightarrow ds \in ?tree)$ **by** *auto*
ultimately
have $\exists c. \text{wf-infpath}\ c \wedge (\forall n. c\ n \in ?tree)$ **using** *konig*[of *?tree*] **by** *blast*
then have $\exists G. \text{wf-infpath}\ G \wedge (\forall n. \neg \text{falsifies}_{cs} (G\ n)\ Cs)$ **by** *auto*
— Apply above infpath lemma:
then show $\exists G. \text{eval}_{cs}\ \text{HFun}\ G\ Cs$ **using** *infpath-model finite-cs* **by** *auto*
qed

lemma *shorter-falsifies_l*:
assumes *falsifies_l* (*ds@d*) *l*
assumes *nat-from-fatomb* (*get-atom l*) < *length ds*
shows *falsifies_l* *ds l*
proof —
let *?i* = *nat-from-fatomb* (*get-atom l*)
from *assms* **have** *i-p*: *ground_l l* \wedge *?i* < *length (ds@d)* \wedge (*ds@d*) ! *?i* = (\neg *sign*
l) **unfolding** *falsifies_l-def* **by** *meson*
moreover
then have *?i* < *length ds* **using** *assms* **by** *auto*
moreover
then have *ds* ! *?i* = (\neg *sign l*) **using** *i-p nth-append*[of *ds d* *?i*] **by** *auto*
ultimately show *?thesis* **using** *assms* **unfolding** *falsifies_l-def* **by** *simp*
qed

theorem *herbrand'-contra*:
assumes *finite-cs*: *finite Cs* $\forall C \in Cs. \text{finite } C$
assumes *unsat*: $\forall G. \neg \text{eval}_{cs}\ \text{HFun}\ G\ Cs$
shows $\exists T. \forall G. \text{branch}\ G\ T \longrightarrow \text{closed-branch}\ G\ T\ Cs$
proof —
from *finite-cs unsat* **have** $\forall T. \exists G. \text{open-branch}\ G\ T\ Cs \implies \exists G. \text{eval}_{cs}\ \text{HFun}\ G\ Cs$
using *herbrand'-contra*[of *Cs*] **by** *blast*
then show *?thesis* **using** *unsat* **by** *blast*
qed

theorem *herbrand*:
assumes *unsat*: $\forall G. \neg \text{eval}_{cs}\ \text{HFun}\ G\ Cs$
assumes *finite-cs*: *finite Cs* $\forall C \in Cs. \text{finite } C$
shows $\exists T. \text{closed-tree}\ T\ Cs$
proof —
from *unsat finite-cs* **obtain** *T* **where** *anybranch T* ($\lambda b. \text{closed-branch}\ b\ T\ Cs$)
using *herbrand'-contra*[of *Cs*] **by** *blast*
then have $\exists T. \text{anybranch}\ T\ (\lambda p. \text{falsifies}_{cs}\ p\ Cs) \wedge \text{anyinternal}\ T\ (\lambda p. \neg \text{falsifies}_{cs}\ p\ Cs)$
using *cutoff-branch-internal*[of *T* $\lambda p. \text{falsifies}_{cs}\ p\ Cs$] **by** *blast*
then show *?thesis* **unfolding** *closed-tree-def* **by** *auto*
qed

end

16 Lifting Lemma

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

locale *unification* =

assumes *unification*: $\bigwedge \sigma L. \text{finite } L \implies \text{unifier}_{l_s} \sigma L \implies \exists \vartheta. \text{mgu}_{l_s} \vartheta L$

begin

A proof of this assumption is available [5] in the IsaFoL project [2]. It uses a similar theorem from the IsaFoR [8] project.

lemma *lifting*:

assumes *fin*: $\text{finite } C \wedge \text{finite } D$

assumes *apart*: $\text{vars}_{l_s} C \cap \text{vars}_{l_s} D = \{\}$

assumes *inst₁*: $\text{instance-of}_{l_s} C' C$

assumes *inst₂*: $\text{instance-of}_{l_s} D' D$

assumes *appl*: $\text{applicable } C' D' L' M' \sigma$

shows $\exists L M \tau. \text{applicable } C D L M \tau \wedge$
 $\text{instance-of}_{l_s} (\text{resolution } C' D' L' M' \sigma) (\text{resolution } C D L M \tau)$

proof –

let $?C'_1 = C' - L'$

let $?D'_1 = D' - M'$

from *inst₁* **obtain** *lmbd* **where** $\text{lmbd-p}: C \cdot_{l_s} \text{lmbd} = C'$ **unfolding** *instance-of_{l_s}*-def

by *auto*

from *inst₂* **obtain** μ **where** $\mu\text{-p}: D \cdot_{l_s} \mu = D'$ **unfolding** *instance-of_{l_s}*-def

by *auto*

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

from $\eta\text{-p}$ **have** $\exists L \subseteq C. L \cdot_{l_s} \eta = L' \wedge (C - L) \cdot_{l_s} \eta = ?C'_1$ **using** *appl*

project-sub[of $\eta C C' L'$] **unfolding** *applicable-def* **by** *auto*

then obtain L **where** $L\text{-p}: L \subseteq C \wedge L \cdot_{l_s} \eta = L' \wedge (C - L) \cdot_{l_s} \eta = ?C'_1$ **by**

auto

let $?C_1 = C - L$

from $\eta\text{-p}$ **have** $\exists M \subseteq D. M \cdot_{l_s} \eta = M' \wedge (D - M) \cdot_{l_s} \eta = ?D'_1$ **using** *appl*

project-sub[of $\eta D D' M'$] **unfolding** *applicable-def* **by** *auto*

then obtain M **where** $M\text{-p}: M \subseteq D \wedge M \cdot_{l_s} \eta = M' \wedge (D - M) \cdot_{l_s} \eta = ?D'_1$

by *auto*

let $?D_1 = D - M$

from *appl* **have** $\text{mgu}_{l_s} \sigma (L' \cup M^C)$ **unfolding** *applicable-def* **by** *auto*

then have $\text{mgu}_{l_s} \sigma ((L \cdot_{l_s} \eta) \cup (M \cdot_{l_s} \eta)^C)$ **using** $L\text{-p}$ $M\text{-p}$ **by** *auto*

then have $\text{mgu}_{l_s} \sigma ((L \cup M^C) \cdot_{l_s} \eta)$ **using** *compls-subls* *subls-union* **by** *auto*

then have $\text{unifier}_{l_s} \sigma ((L \cup M^C) \cdot_{l_s} \eta)$ **unfolding** *mgu_{l_s}*-def

by *auto*

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

— Showing that we have the desired resolvent:

let $?E = ((C - L) \cup (D - M)) \cdot_{l_s} \tau$
have $?E \cdot_{l_s} \varphi = (?C_1 \cup ?D_1) \cdot_{l_s} (\tau \cdot \varphi)$ **using** $subls-union\ composition-conseq2ls$
by *auto*
also have $\dots = (?C_1 \cup ?D_1) \cdot_{l_s} (\eta \cdot \sigma)$ **using** φ -p **by** *auto*
also have $\dots = ((?C_1 \cdot_{l_s} \eta) \cup (?D_1 \cdot_{l_s} \eta)) \cdot_{l_s} \sigma$ **using** $subls-union\ composition-conseq2ls$
by *auto*
also have $\dots = (?C'_1 \cup ?D'_1) \cdot_{l_s} \sigma$ **using** η -p *L-p M-p* **by** *auto*
finally have $?E \cdot_{l_s} \varphi = ((C' - L') \cup (D' - M')) \cdot_{l_s} \sigma$ **by** *auto*
then have $inst$: $instance-of_{l_s} (resolution\ C'\ D'\ L'\ M'\ \sigma) (resolution\ C\ D\ L\ M\ \tau)$
unfolding $resolution-def\ instance-of_{l_s}$ -def **by** *blast*

— Showing that the resolution is applicable:

{
 have $C' \neq \{\}$ **using** $appl$ **unfolding** $applicable-def$ **by** *auto*
 then have $C \neq \{\}$ **using** η -p **by** *auto*
} **moreover** {
 have $D' \neq \{\}$ **using** $appl$ **unfolding** $applicable-def$ **by** *auto*
 then have $D \neq \{\}$ **using** η -p **by** *auto*
} **moreover** {
 have $L' \neq \{\}$ **using** $appl$ **unfolding** $applicable-def$ **by** *auto*
 then have $L \neq \{\}$ **using** L -p **by** *auto*
} **moreover** {
 have $M' \neq \{\}$ **using** $appl$ **unfolding** $applicable-def$ **by** *auto*
 then have $M \neq \{\}$ **using** M -p **by** *auto*
}
ultimately have $apll$: $applicable\ C\ D\ L\ M\ \tau$
using $apart\ L$ -p M -p τ -p **unfolding** $applicable-def$ **by** *auto*

from $inst\ apll$ **show** $?thesis$ **by** *auto*
qed

17 Completeness

lemma $falsifies_g$ -empty:
 assumes $falsifies_g \ \square\ C$
 shows $C = \{\}$
proof —
 have $\forall l \in C. False$
 proof
 fix l

```

    assume  $l \in C$ 
    then have  $\text{falsifies}_l \ [] \ l$  using assms by auto
    then show False unfolding falsifiesl-def by (cases l) auto
  qed
  then show ?thesis by auto
qed

lemma falsifiescs-empty:
  assumes  $\text{falsifies}_c \ [] \ C$ 
  shows  $C = \{\}$ 
proof -
  from assms obtain  $C'$  where  $C'-p$ : instance-ofls  $C' \ C \wedge \text{falsifies}_g \ [] \ C'$  by
auto
  then have  $C' = \{\}$  using falsifiesg-empty by auto
  then show  $C = \{\}$  using  $C'-p$  unfolding instance-ofls-def by auto
qed

lemma complements-do-not-falsify':
  assumes  $l1C1'$ :  $l_1 \in C_1'$ 
  assumes  $l2C1'$ :  $l_2 \in C_1'$ 
  assumes comp:  $l_1 = l_2^c$ 
  assumes falsif:  $\text{falsifies}_g \ G \ C_1'$ 
  shows False
proof (cases l1)
  case (Pos p ts)
  let  $?i1 = \text{nat-from-fatom } (p, ts)$ 

  from assms have gr:  $\text{ground}_l \ l_1$  unfolding falsifiesl-def by auto
  then have  $\text{Neg}: l_2 = \text{Neg } p \ ts$  using comp Pos by (cases l2) auto

  from falsif have  $\text{falsifies}_l \ G \ l_1$  using  $l1C1'$  by auto
  then have  $G \ ! \ ?i1 = \text{False}$  using  $l1C1' \text{Pos}$  unfolding falsifiesl-def by (induction
Pos p ts) auto
  moreover
  let  $?i2 = \text{nat-from-fatom } (\text{get-atom } l_2)$ 
  from falsif have  $\text{falsifies}_l \ G \ l_2$  using  $l2C1'$  by auto
  then have  $G \ ! \ ?i2 = (\neg \text{sign } l_2)$  unfolding falsifiesl-def by meson
  then have  $G \ ! \ ?i1 = (\neg \text{sign } l_2)$  using Pos Neg comp by simp
  then have  $G \ ! \ ?i1 = \text{True}$  using Neg by auto
  ultimately show ?thesis by auto
next
  case (Neg p ts)
  let  $?i1 = \text{nat-from-fatom } (p, ts)$ 

  from assms have gr:  $\text{ground}_l \ l_1$  unfolding falsifiesl-def by auto
  then have  $\text{Pos}: l_2 = \text{Pos } p \ ts$  using comp Neg by (cases l2) auto

  from falsif have  $\text{falsifies}_l \ G \ l_1$  using  $l1C1'$  by auto
  then have  $G \ ! \ ?i1 = \text{True}$  using  $l1C1' \text{Neg}$  unfolding falsifiesl-def by (metis

```


get-atom.simps(2) literal.disc(2)
moreover
let $?i2 = \text{nat-from-fatomb} (\text{get-atom } l_2)$
from *falsif* **have** $\text{falsifies}_1 G l_2$ **using** $l_2 C_1'$ **by** *auto*
then **have** $G ! ?i2 = (\neg \text{sign } l_2)$ **unfolding** *falsifies₁-def* **by** *meson*
then **have** $G ! ?i1 = (\neg \text{sign } l_2)$ **using** *Pos Neg comp* **by** *simp*
then **have** $G ! ?i1 = \text{False}$ **using** *Pos* **using** *literal.disc(1)* **by** *blast*
ultimately **show** *?thesis* **by** *auto*
qed

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

lemma *other-falsified*:
assumes $C_1'-p$: $\text{ground}_{l_s} C_1' \wedge \text{falsifies}_g (B@[d]) C_1'$
assumes $l-p$: $l \in C_1'$ *nat-from-fatomb* ($\text{get-atom } l$) = *length* B
assumes *other*: $lo \in C_1'$ $lo \neq l$
shows $\text{falsifies}_1 B lo$

proof –
let $?i = \text{nat-from-fatomb} (\text{get-atom } lo)$
have $\text{ground-}l_2$: $\text{ground}_1 l$ **using** $l-p C_1'-p$ **by** *auto*
– They are, of course, also ground:
have $\text{ground-}lo$: $\text{ground}_1 lo$ **using** $C_1'-p$ *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** loB_2 : $\text{falsifies}_1 (B@[d]) lo$ **using** *other* **by** *auto*
then **have** $?i < \text{length} (B @ [d])$ **unfolding** *falsifies₁-def* **by** *meson*
– And they have numbers in the range of $B @ [d]$, i.e. less than *length* $B + 1$:
then **have** $\text{nat-from-fatomb} (\text{get-atom } lo) < \text{length } B + 1$ **using** *unddiag-diag-fatomb*
by (*cases lo*) *auto*
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 $lc-lo$: $lo \neq l^c$ **using** $C_1'-p$ $l-p$ *other complements-do-not-falsify* [*of lo C₁' l (B@[d])*] **by** *auto*
from $l-lo$ $lc-lo$ **have** $\text{get-atom } l \neq \text{get-atom } lo$ **using** *sign-comp-atom* **by** *metis*
then **have** $\text{nat-from-fatomb} (\text{get-atom } lo) \neq \text{nat-from-fatomb} (\text{get-atom } l)$
using *nat-from-fatomb-bij ground-lo ground-}l_2 ground₁-ground-fatomb*
unfolding *bij-betw-def inj-on-def* **by** *metis*
– Therefore they have different numbers:
then **have** $\text{nat-from-fatomb} (\text{get-atom } lo) \neq \text{length } B$ **using** $l-p$ **by** *auto*
ultimately
– So their numbers are in the range of B :
have $\text{nat-from-fatomb} (\text{get-atom } lo) < \text{length } B$ **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** $\text{lo}B_2$ **shorter-falsifies** **by** *blast*
qed

theorem *completeness'*:

shows $\text{closed-tree } T \text{ } Cs \implies \forall C \in Cs. \text{ finite } C \implies \exists Cs'. \text{ resolution-deriv } Cs \text{ } Cs' \wedge \{\} \in Cs'$

proof (*induction* T *arbitrary*: Cs *rule*: *measure-induct-rule*[*of treesize*])

fix $T::\text{tree}$

fix $Cs :: \text{fterm clause set}$

assume *ih*: $(\bigwedge T' \text{ } Cs. \text{ treesize } T' < \text{ treesize } T \implies \text{ closed-tree } T' \text{ } Cs \implies \forall C \in Cs. \text{ finite } C \implies$

$\exists Cs'. \text{ resolution-deriv } Cs \text{ } Cs' \wedge \{\} \in Cs')$

assume *clo*: $\text{closed-tree } T \text{ } Cs$

assume *finite-Cs*: $\forall C \in Cs. \text{ finite } C$

{ — Base case:

assume $\text{treesize } T = 0$

then have $T = \text{Leaf}$ **using** *treesize-Leaf* **by** *auto*

then have $\text{closed-branch } [] \text{ Leaf } Cs$ **using** *branch-inv-Leaf clo unfolding*

closed-tree-def **by** *auto*

then have $\text{falsifies}_{Cs} [] \text{ } Cs$ **by** *auto*

then have $\{\} \in Cs$ **using** *falsifies_{Cs}-empty* **by** *auto*

then have $\exists Cs'. \text{ resolution-deriv } Cs \text{ } Cs' \wedge \{\} \in Cs'$ **unfolding** *resolution-deriv-def*

by *auto*

}

moreover

{ — Induction case:

assume $\text{treesize } T > 0$

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

— Finding sibling branches and their corresponding clauses:

then obtain B **where** $b\text{-}p: \text{ internal } B \text{ } T \wedge \text{ branch } (B@[True]) \text{ } T \wedge \text{ branch } (B@[False]) \text{ } T$

using *internal-branch*[*of* - [] - T] *Branching-Leaf-Leaf-Tree* **by** *fastforce*

let $?B_1 = B@[True]$

let $?B_2 = B@[False]$

obtain C_{1o} **where** $C_{1o}\text{-}p: C_{1o} \in Cs \wedge \text{ falsifies}_c ?B_1 \text{ } C_{1o}$ **using** $b\text{-}p$ *clo unfolding* *closed-tree-def* **by** *metis*

obtain C_{2o} **where** $C_{2o}\text{-}p: C_{2o} \in Cs \wedge \text{ falsifies}_c ?B_2 \text{ } C_{2o}$ **using** $b\text{-}p$ *clo unfolding* *closed-tree-def* **by** *metis*

— Standardizing the clauses apart:

let $?C_1 = \text{std}_1 \text{ } C_{1o}$

let $?C_2 = \text{std}_2 \text{ } C_{2o}$

have $C_{1\text{-}p}: \text{ falsifies}_c ?B_1 \text{ } ?C_1$ **using** *std₁-falsifies* $C_{1o}\text{-}p$ **by** *auto*

have $C_{2\text{-}p}: \text{ falsifies}_c ?B_2 \text{ } ?C_2$ **using** *std₂-falsifies* $C_{2o}\text{-}p$ **by** *auto*

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

— We go down to the ground world.

— Finding the falsifying ground instance C_1' of $C_1 o \cdot l_s (\lambda x. \varepsilon ("1" @ x))$, and proving properties about it:

— C_1' is falsified by $B @ [True]$:

from $C_1\text{-}p$ **obtain** C_1' **where** $C_1'\text{-}p$: *ground* $_{l_s}$ $C_1' \wedge$ *instance-of* $_{l_s}$ $C_1' ?C_1 \wedge$ *falsifies* $_g$? B_1 C_1' **by** *metis*

have \neg *falsifies* $_c$ B $C_1 o$ **using** $C_1 o\text{-}p$ *b-p clo unfolding closed-tree-def* **by** *metis*
then **have** \neg *falsifies* $_c$ B ? C_1 **using** *std1-falsifies* **using** *prod.exhaust-sel* **by** *blast*

— C_1' is not falsified by B :

then **have** $l\text{-}B$: \neg *falsifies* $_g$ B C_1' **using** $C_1'\text{-}p$ **by** *auto*

— C_1' contains a literal l_1 that is falsified by $B @ [True]$, but not B :

from $C_1'\text{-}p$ $l\text{-}B$ **obtain** l_1 **where** $l_1\text{-}p$: $l_1 \in C_1' \wedge$ *falsifies* $_l$ ($B @ [True]$) $l_1 \wedge$ \neg (*falsifies* $_l$ B l_1) **by** *auto*

let ? i = *nat-from-fat* om (*get-atom* l_1)

— l_1 is of course ground:

have *ground-l1*: *ground* $_l$ l_1 **using** $C_1'\text{-}p$ $l_1\text{-}p$ **by** *auto*

from $l_1\text{-}p$ **have** \neg (? i < *length* $B \wedge B ! ?i = (\neg$ *sign* $l_1))$ **using** *ground-l1*
unfolding *falsifies* $_l\text{-}def$ **by** *meson*

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

moreover

from $l_1\text{-}p$ **have** ? i < *length* ($B @ [True]$) \wedge ($B @ [True]) ! ?i = (\neg$ *sign* $l_1)$
unfolding *falsifies* $_l\text{-}def$ **by** *meson*

ultimately

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

— l_1 is negative:

from $l_1\text{-}sign\text{-}no$ **have** $l_1\text{-}sign$: *sign* $l_1 = False$ **by** *auto*

from $l_1\text{-}sign\text{-}no$ **have** $l_1\text{-}no$: *nat-from-fat* om (*get-atom* l_1) = *length* B **by** *auto*

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

from $C_1'\text{-}p$ $l_1\text{-}no$ $l_1\text{-}p$ **have** $B\text{-}C_1'l_1$: *falsifies* $_g$ $B (C_1' - \{l_1\})$
using *other-falsified* **by** *blast*

— We do the same exercise for $C_2 o \cdot l_s (\lambda x. \varepsilon ("2" @ x))$, C_2' , $B @ [False]$, l_2 :

from $C_2\text{-}p$ **obtain** C_2' **where** $C_2'\text{-}p$: *ground* $_{l_s}$ $C_2' \wedge$ *instance-of* $_{l_s}$ $C_2' ?C_2 \wedge$ *falsifies* $_g$? B_2 C_2' **by** *metis*

have \neg *falsifies* $_c$ B $C_2 o$ **using** $C_2 o\text{-}p$ *b-p clo unfolding closed-tree-def* **by** *metis*

then have $\neg \text{falsifies}_c B \ ?C_2$ **using** *std₂-falsifies* **using** *prod.exhaust-sel* **by** *blast*

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

— C_2' contains a literal l_2 that is falsified by $B @ [False]$, but not B :

from $C_2'\text{-}p \ l\text{-}B$ **obtain** l_2 **where** $l_2\text{-}p: l_2 \in C_2' \wedge \text{falsifies}_l (B@[False]) \ l_2 \wedge \neg \text{falsifies}_l B \ l_2$ **by** *auto*

let $?i = \text{nat-from-fatom} (\text{get-atom } l_2)$

have $\text{ground-}l_2: \text{ground}_l \ l_2$ **using** $C_2'\text{-}p \ l_2\text{-}p$ **by** *auto*

from $l_2\text{-}p$ **have** $\neg (?i < \text{length } B \wedge B ! ?i = (\neg \text{sign } l_2))$ **using** $\text{ground-}l_2$ **unfolding** *falsifies_l-def* **by** *meson*

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

moreover

from $l_2\text{-}p$ **have** $?i < \text{length } (B @ [False]) \wedge (B @ [False]) ! ?i = (\neg \text{sign } l_2)$ **unfolding** *falsifies_l-def* **by** *meson*

ultimately

have $l_2\text{-sign-no}: ?i = \text{length } B \wedge (B @ [False]) ! ?i = (\neg \text{sign } l_2)$ **by** *auto*

— l_2 is negative:

from $l_2\text{-sign-no}$ **have** $l_2\text{-sign}: \text{sign } l_2 = \text{True}$ **by** *auto*

from $l_2\text{-sign-no}$ **have** $l_2\text{-no}: \text{nat-from-fatom} (\text{get-atom } l_2) = \text{length } B$ **by** *auto*

— All the other literals in C_2' must be falsified by B , since they are falsified by $B @ [False]$, but not l_2 .

from $C_2'\text{-}p \ l_2\text{-no} \ l_2\text{-}p$ **have** $B\text{-}C_2'l_2: \text{falsifies}_g B (C_2' - \{l_2\})$ **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 $l_1\text{-no} \ l_2\text{-no} \ \text{ground-}l_1 \ \text{ground-}l_2$ **have** $\text{get-atom } l_1 = \text{get-atom } l_2$

using *nat-from-fatom-bij* *ground_l-ground-fatom*

unfolding *bij-betw-def* *inj-on-def* **by** *metis*

then show $l_2^c = l_1$ **using** $l_1\text{-sign} \ l_2\text{-sign}$ **using** *sign-comp-atom* **by** *metis*

qed

have *applicable* $C_1' \ C_2' \ \{l_1\} \ \{l_2\}$ *Resolution.ε* **unfolding** *applicable-def*

using $l_1\text{-}p \ l_2\text{-}p \ C_1'\text{-}p \ \text{ground}_{l_s}\text{-vars}_{l_s} \ l_2\text{cisl}_1 \ \text{empty-comp2}$ **unfolding** *mgul_s-def unifier_{l_s}-def* **by** *auto*

— Lifting to get a resolvent of $C_1 \circ \cdot_{l_s} (\lambda x. \varepsilon ("1" @ x))$ and $C_2 \circ \cdot_{l_s} (\lambda x. \varepsilon ("2" @ x))$:

then obtain $L_1 \ L_2 \ \tau$ **where** $L_1 L_2 \tau\text{-}p: \text{applicable} \ ?C_1 \ ?C_2 \ L_1 \ L_2 \ \tau \wedge \text{instance-of}_{l_s} (\text{resolution } C_1' \ C_2' \ \{l_1\} \ \{l_2\} \ \text{Resolution.}\varepsilon) (\text{resolution } ?C_1 \ ?C_2 \ L_1 \ L_2 \ \tau)$

using *std-apart-apart* $C_1'\text{-}p \ C_2'\text{-}p$ *lifting*[of $?C_1 \ ?C_2 \ C_1' \ C_2' \ \{l_1\} \ \{l_2\}$]

Resolution.ε] *fin* **by** *auto*

- Defining the clause to be derived, the new clausal form and the new tree:
- We name the resolvent C .

obtain C **where** C - p : $C = \text{resolution } ?C_1 \ ?C_2 \ L_1 \ L_2 \ \tau$ **by** *auto*

obtain $CsNext$ **where** $CsNext$ - p : $CsNext = Cs \cup \{?C_1, ?C_2, C\}$ **by** *auto*

obtain T'' **where** T'' - p : $T'' = \text{delete } B \ T$ **by** *auto*

- Here we delete the two branch children $B @ [True]$ and $B @ [False]$ of B .

- Our new clause is falsified by the branch B of our new tree:

have *falsifies_g* $B ((C_1' - \{l_1\}) \cup (C_2' - \{l_2\}))$ **using** B - $C_1'l_1$ B - $C_2'l_2$ **by** *cases auto*

then have *falsifies_g* $B (\text{resolution } C_1' \ C_2' \ \{l_1\} \ \{l_2\} \ \text{Resolution.}\epsilon)$ **unfolding** *resolution-def empty-subls* **by** *auto*

then have *falsifies-C*: *falsifies_c* $B \ C$ **using** C - $p \ L_1L_2\tau$ - p **by** *auto*

have T'' -*smaller*: *treesize* $T'' < \text{treesize } T$ **using** *treezise-delete* T'' - $p \ b$ - p **by** *auto*

have T'' -*bran*: *anybranch* $T'' (\lambda b. \text{closed-branch } b \ T'' \ CsNext)$

proof (*rule allI*; *rule impI*)

fix b

assume br : *branch* $b \ T''$

from br **have** $b = B \vee \text{branch } b \ T$ **using** *branch-delete* T'' - p **by** *auto*

then show *closed-branch* $b \ T'' \ CsNext$

proof

assume $b=B$

then show *closed-branch* $b \ T'' \ CsNext$ **using** *falsifies-C* $br \ CsNext$ - p **by**

auto

next

assume *branch* $b \ T$

then show *closed-branch* $b \ T'' \ CsNext$ **using** *clo* $br \ T''$ - $p \ CsNext$ - p

unfolding *closed-tree-def* **by** *auto*

qed

qed

then have T'' -*bran2*: *anybranch* $T'' (\lambda b. \text{falsifies}_{cs} \ b \ CsNext)$ **by** *auto*

- We cut the tree even smaller to ensure only the branches are falsified, i.e. it is a closed tree:

obtain T' **where** T' - p : $T' = \text{cutoff } (\lambda G. \text{falsifies}_{cs} \ G \ CsNext) \ [] \ T''$ **by** *auto*

have T' -*smaller*: *treesize* $T' < \text{treesize } T$ **using** *treesize-cutoff*[*of* $\lambda G. \text{falsifies}_{cs} \ G \ CsNext \ [] \ T''$] T'' -*smaller* **unfolding** T' - p **by** *auto*

from T'' -*bran2* **have** *anybranch* $T' (\lambda b. \text{falsifies}_{cs} \ b \ CsNext)$ **using** *cutoff-branch*[*of* $T'' \ \lambda b. \text{falsifies}_{cs} \ b \ CsNext$] T' - p **by** *auto*

then have T' -*bran*: *anybranch* $T' (\lambda b. \text{closed-branch } b \ T' \ CsNext)$ **by** *auto*

have T' -*intr*: *anyinternal* $T' (\lambda p. \neg \text{falsifies}_{cs} \ p \ CsNext)$ **using** T' - p *cutoff-internal*[*of* $T'' \ \lambda b. \text{falsifies}_{cs} \ b \ CsNext$] T'' -*bran2* **by** *blast*

have T' -*closed*: *closed-tree* $T' \ CsNext$ **using** T' -*bran* T' -*intr* **unfolding**

closed-tree-def **by** *auto*

have *finite-CsNext*: $\forall C \in CsNext. \text{finite } C$ **unfolding** *CsNext-p C-p resolution-def*
using *finite-Cs fin* **by** *auto*

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

from *T'-smaller T'-closed* **have** $\exists Cs''. \text{resolution-deriv } CsNext \ Cs'' \wedge \{\} \in Cs''$ **using** *ih[of T' CsNext] finite-CsNext* **by** *blast*

then obtain *Cs''* **where** $Cs''\text{-p}: \text{resolution-deriv } CsNext \ Cs'' \wedge \{\} \in Cs''$ **by** *auto*

moreover

{ — Proving that we can actually derive the new clausal form:

have *resolution-step Cs (Cs \cup $\{?C_1\})$* **using** *std₁-renames standardize-apart C₁o-p* **by** (*metis Un-insert-right*)

moreover

have *resolution-step (Cs \cup $\{?C_1\}) (Cs \cup \{?C_1\} \cup \{?C_2\})$* **using** *std₂-renames[of C₂o] standardize-apart[of C₂o - ?C₂] C₂o-p* **by** *auto*

then have *resolution-step (Cs \cup $\{?C_1\}) (Cs \cup \{?C_1, ?C_2\})$* **by** (*simp add: insert-commute*)

moreover

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

using *L₁L₂ τ -p resolution-rule[of ?C₁ Cs \cup $\{?C_1, ?C_2\}$?C₂ L₁ L₂ τ]* **using** *C-p* **by** *auto*

then have *resolution-step (Cs \cup $\{?C_1, ?C_2\}) CsNext$* **using** *CsNext-p* **by** (*simp add: Un-commute*)

ultimately

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

}

— Combining the two derivations, we get the desired derivation from *Cs* of $\{\}$:

ultimately have *resolution-deriv Cs Cs''* **unfolding** *resolution-deriv-def* **by** *auto*

then have $\exists Cs'. \text{resolution-deriv } Cs \ Cs' \wedge \{\} \in Cs'$ **using** *Cs''-p* **by** *auto*

}

ultimately show $\exists Cs'. \text{resolution-deriv } Cs \ Cs' \wedge \{\} \in Cs'$ **by** *auto*

qed

theorem completeness:

assumes *finite-cs: finite Cs $\forall C \in Cs. \text{finite } C$*

assumes *unsat: $\forall (F::\text{hterm fun-denot}) (G::\text{hterm pred-denot}). \neg \text{eval}_{cs} F G Cs$*

shows $\exists Cs'. \text{resolution-deriv } Cs \ Cs' \wedge \{\} \in Cs'$

proof —

from *unsat* **have** $\forall (G::\text{hterm pred-denot}). \neg \text{eval}_{cs} HFun G Cs$ **by** *auto*

then obtain *T* **where** *closed-tree T Cs* **using** *herbrand assms* **by** *blast*

then show $\exists Cs'. \text{resolution-deriv } Cs \ Cs' \wedge \{\} \in Cs'$ **using** *completeness' assms*
by *auto*

qed

end — unification locale

end

18 Examples

theory *Examples* imports *Resolution* begin

```
value Var "x"
value Fun "one" []
value Fun "mul" [Var "y", Var "y"]
value Fun "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []]

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

fun $F_{nat} :: nat \text{ fun-denot}$ where

```
 $F_{nat} f [n, m] =$ 
  (if  $f = \text{"add"}$  then  $n + m$  else
   if  $f = \text{"mul"}$  then  $n * m$  else 0)
|  $F_{nat} f [] =$ 
  (if  $f = \text{"one"}$  then 1 else
   if  $f = \text{"zero"}$  then 0 else 0)
|  $F_{nat} f us = 0$ 
```

fun $G_{nat} :: nat \text{ pred-denot}$ where

```
 $G_{nat} p [x, y] =$ 
  (if  $p = \text{"less"} \wedge x < y$  then True else
   if  $p = \text{"greater"} \wedge x > y$  then True else
   if  $p = \text{"equals"} \wedge x = y$  then True else False)
|  $G_{nat} p us = False$ 
```

fun $E_{nat} :: nat \text{ var-denot}$ where

```
 $E_{nat} x =$ 
  (if  $x = \text{"x"}$  then 26 else
   if  $x = \text{"y"}$  then 5 else 0)
```

lemma $eval_t E_{nat} F_{nat} (Var \text{"x"}) = 26$

by auto

lemma $eval_t E_{nat} F_{nat} (Fun \text{"one"} []) = 1$

by auto

lemma $eval_t E_{nat} F_{nat} (Fun \text{"mul"} [Var \text{"y"}, Var \text{"y"}]) = 25$

by auto

lemma

$eval_t E_{nat} F_{nat} (Fun \text{"add"} [Fun \text{"mul"} [Var \text{"y"}, Var \text{"y"}], Fun \text{"one"} []) = 26$

by auto

lemma $eval_l E_{nat} F_{nat} G_{nat} (Pos \text{ "greater" } [Var \text{ "x"}, Var \text{ "y"}]) = True$
by auto

lemma $eval_l E_{nat} F_{nat} G_{nat} (Neg \text{ "less" } [Var \text{ "x"}, Var \text{ "y"}]) = True$
by auto

lemma $eval_l E_{nat} F_{nat} G_{nat} (Pos \text{ "less" } [Var \text{ "x"}, Var \text{ "y"}]) = False$
by auto

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

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

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

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

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

theorem $empty-mgu: unifier_{l_s} \varepsilon L \implies mgu_{l_s} \varepsilon L$

unfolding $unifier_{l_s}\text{-def}$ $mgu_{l_s}\text{-def}$ **apply auto**

apply $(rule\text{-tac } x=u \text{ in } exI)$

using $empty\text{-comp1}$ $empty\text{-comp2}$ **apply auto**

done

theorem $unifier\text{-single}: unifier_{l_s} \sigma \{l\}$

unfolding $unifier_{l_s}\text{-def}$ **by auto**

theorem $resolution\text{-rule}'$:

$C_1 \in Cs \implies C_2 \in Cs \implies applicable C_1 C_2 L_1 L_2 \sigma$

$\implies C = \{resolution C_1 C_2 L_1 L_2 \sigma\}$

$\implies resolution\text{-step } Cs (Cs \cup C)$

using $resolution\text{-rule}$ **by auto**

lemma $resolution\text{-example1}$:

$resolution\text{-deriv} \{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}$
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}, \{\}\}$

proof –

have $resolution\text{-step}$

$\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}$

$(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\} \cup \{\{NP\}\})$

apply $(rule resolution\text{-rule}'[of \{NP, PQ\} - \{NQ\} \{PQ\} \{NQ\} \varepsilon])$

unfolding $applicable\text{-def}$ $vars_{l_s}\text{-def}$ $vars_l\text{-def}$

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

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

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

definition *Pb* :: *fterm literal* **where**
 $Pb = Pos \ "b" \ []$

definition $Nb :: fterm\ literal$ **where**

$Nb = Neg\ "b" []$

definition $Paa :: fterm\ literal$ **where**

$Paa = Pos\ "a" [Fun\ "a" []]$

definition $Naa :: fterm\ literal$ **where**

$Naa = Neg\ "a" [Fun\ "a" []]$

definition $Pax :: fterm\ literal$ **where**

$Pax = Pos\ "a" [Var\ "x"]$

definition $Nax :: fterm\ literal$ **where**

$Nax = Neg\ "a" [Var\ "x"]$

definition $mguPaaPax :: substitution$ **where**

$mguPaaPax = (\lambda x. if\ x = "x" then\ Fun\ "a" [] else\ Var\ x)$

lemma $mguPaaPax-mgu: mgu_{l_s}\ mguPaaPax\ \{Paa, Pax\}$

proof –

let $?\sigma = \lambda x. if\ x = "x" then\ Fun\ "a" [] else\ Var\ x$

have $a: unifier_{l_s}\ (\lambda x. if\ x = "x" then\ Fun\ "a" [] else\ Var\ x)\ \{Paa, Pax\}$ **un-**
folding $Paa-def\ Pax-def\ unifier_{l_s}-def$ **by** $auto$

have $b: \forall u. unifier_{l_s}\ u\ \{Paa, Pax\} \longrightarrow (\exists i. u = ?\sigma \cdot i)$

proof $(rule; rule)$

fix u

assume $unifier_{l_s}\ u\ \{Paa, Pax\}$

then **have** $uuu: u\ "x" = Fun\ "a" []$ **unfolding** $unifier_{l_s}-def\ Paa-def\ Pax-def$

by $auto$

have $?\sigma \cdot u = u$

proof

fix x

{

assume $x = "x"$

moreover

have $(?\sigma \cdot u)\ "x" = Fun\ "a" []$ **unfolding** $composition-def$ **by** $auto$

ultimately **have** $(?\sigma \cdot u)\ x = u\ x$ **using** uuu **by** $auto$

}

moreover

{

assume $x \neq "x"$

then **have** $(?\sigma \cdot u)\ x = (\varepsilon\ x) \cdot_t u$ **unfolding** $composition-def$ **by** $auto$

then **have** $(?\sigma \cdot u)\ x = u\ x$ **by** $auto$

}

ultimately **show** $(?\sigma \cdot u)\ x = u\ x$ **by** $auto$

qed

then **have** $\exists i. ?\sigma \cdot i = u$ **by** $auto$

then **show** $\exists i. u = ?\sigma \cdot i$ **by** $auto$

```

qed
from a b show ?thesis unfolding mgu1s-def unfolding mguPaaPax-def by
auto
qed

lemma resolution-example2:
  resolution-deriv  $\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}\}$ 
     $\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}, \{Na, Pb\}, \{Na\}, \{\}\}$ 

proof –
  have resolution-step
     $\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}\}$ 
     $(\{\{Nb, Na\}, \{Pax\}, \{Pa\}, \{Na, Pb, Naa\}\} \cup \{\{Na, Pb\}\})$ 
  apply (rule resolution-rule'of  $\{Pax\} - \{Na, Pb, Naa\}$   $\{Pax\}$   $\{Naa\}$  mguPaaPax
  ])
    using mguPaaPax-mgu unfolding applicable-def vars1s-def vars1-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  $\{Nb, Na\} - \{Na, Pb\}$   $\{Nb\}$   $\{Pb\}$   $\epsilon$ )
  unfolding applicable-def vars1s-def vars1-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  $\{Na\} - \{Pa\}$   $\{Na\}$   $\{Pa\}$   $\epsilon$ )
  unfolding applicable-def vars1s-def vars1-def
    Pa-def Nb-def Na-def PP-def resolution-def
  using unifier-single empty-mgu apply auto
done

```

```

then have resolution-step
  {{{Nb,Na},{Pax},{Pa},{Na,Pb,Naa},{Na,Pb},{Na}}}
  ({{{Nb,Na},{Pax},{Pa},{Na,Pb,Naa},{Na,Pb},{Na},{}}}
  by (simp add: insert-commute)
ultimately
have resolution-deriv {{{Nb,Na},{Pax},{Pa},{Na,Pb,Naa}}}
  {{{Nb,Na},{Pax},{Pa},{Na,Pb,Naa},{Na,Pb},{Na},{}}}
  unfolding resolution-deriv-def by auto
then show ?thesis by auto
qed

lemma ref-sound:
  assumes deriv: resolution-deriv Cs Cs'  $\wedge$  {}  $\in$  Cs'
  shows  $\neg$ evalCs F G Cs
proof –
  from deriv have evalCs F G Cs  $\implies$  evalCs F G Cs' using lsound-derivation by
  auto
  moreover
  from deriv have evalCs F G Cs'  $\implies$  evalC F G {} unfolding evalCs-def by
  auto
  moreover
  then have evalC F G {}  $\implies$  False unfolding evalC-def by auto
  ultimately show ?thesis by auto
qed

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

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

end

```

References

- [1] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 3rd edition, 2012.
- [2] J. C. Blanchette, M. Fleury, A. Schlichtkrull, and D. Traytel. IsaFoL: Isabelle Formalization of Logic. https://bitbucket.org/jasmin_blanchette/isafol.
- [3] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1st edition, 1973.
- [4] A. Leitsch. *The Resolution Calculus*. Texts in theoretical computer science. Springer, 1997.

- [5] A. Schlichtkrull. Formalization of first-order unordered resolution. https://bitbucket.org/jasmin_blanchette/isafol/src/master/Unordered_Resolution/.
- [6] A. Schlichtkrull. Formalization of resolution calculus in Isabelle. Msc thesis, Technical University of Denmark, 2015. <https://people.compute.dtu.dk/andschl/Thesis.pdf>.
- [7] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In *ITP 2016*, volume 9807 of *LNCS*. Springer, 2016.
- [8] C. Sternagel and R. Thiemann. An Isabelle/HOL formalization of rewriting for certified termination analysis. <http://cl-informatik.uibk.ac.at/software/ceta/>.