

The Resolution Calculus for First-Order Logic

Anders Schlichtkrull

August 16, 2018

Abstract

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

Contents

1	Terms and Literals	3
1.1	Ground	3
1.2	Auxiliary	4
1.3	Conversions	4
1.3.1	Conversions - Terms and Herbrand Terms	4
1.3.2	Conversions - Literals and Herbrand Literals	5
1.3.3	Conversions - Atoms and Herbrand Atoms	6
1.4	Enumerations	7
1.4.1	Enumerating Strings	7
1.4.2	Enumerating Herbrand Atoms	8
1.4.3	Enumerating Ground Atoms	9
2	Trees	9
2.1	Sizes	9
2.2	Paths	10
2.3	Branches	11

2.4	Internal Paths	13
2.5	Deleting Nodes	15
3	Possibly Infinite Trees	23
3.1	Infinite Paths	24
4	König's Lemma	25
5	More Terms and Literals	26
6	Clauses	27
7	Semantics	28
7.1	Semantics of Ground Terms	29
8	Substitutions	29
8.1	The Empty Substitution	30
8.2	Substitutions and Ground Terms	31
8.3	Composition	32
8.4	Merging substitutions	34
8.5	Standardizing apart	36
9	Unifiers	38
9.1	Most General Unifiers	40
10	Resolution	40
11	Soundness	41
12	Herbrand Interpretations	45
13	Partial Interpretations	46
14	Semantic Trees	49
15	Herbrand's Theorem	50
16	Lifting Lemma	56
17	Completeness	57
18	Examples	67
19	The Unification Theorem	73
20	Instance of completeness theorem	78

1 Terms and Literals

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

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

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

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

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

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

fun *get-atom* :: *'t literal* \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-fatom*:
 assumes *ground_l l*
 shows *get-atom l \in ground-fatoms*
 using *assms* **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* ‘ $S = (\text{UNIV} :: \text{nat set})$ ’ **by** *auto*

then show $\neg \text{finite } S$ **by** (*metis finite-imageI infinite-UNIV-nat*)

qed

lemma *inv-into-f-f*:

assumes *bij-betw* $f A B$

assumes $a \in A$

shows $(\text{inv-into } A f) (f a) = a$

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 $f ((\text{inv-into } A f) b) = b$

using *assms bij-betw-inv-into-right* **by** *metis*

1.3 Conversions

1.3.1 Conversions - Terms and Herbrand Terms

fun *fterm-of-hterm* :: $\text{hterm} \Rightarrow \text{fterm}$ **where**

fterm-of-hterm ($\text{HFun } p \text{ ts}$) = $\text{Fun } p (\text{map } \text{fterm-of-hterm } \text{ts})$

definition *fterms-of-hterms* :: $\text{hterm list} \Rightarrow \text{fterm list}$ **where**

fterms-of-hterms $\text{ts} \equiv \text{map } \text{fterm-of-hterm } \text{ts}$

fun *hterm-of-fterm* :: $\text{fterm} \Rightarrow \text{hterm}$ **where**

hterm-of-fterm ($\text{Fun } p \text{ ts}$) = $\text{HFun } p (\text{map } \text{hterm-of-fterm } \text{ts})$

definition *hterms-of-fterms* :: $\text{fterm list} \Rightarrow \text{hterm list}$ **where**

hterms-of-fterms $\text{ts} \equiv \text{map } \text{hterm-of-fterm } \text{ts}$

lemma *hterm-of-fterm-fterm-of-hterm[simp]*: $\text{hterm-of-fterm } (\text{fterm-of-hterm } t) = t$

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

lemma *hterms-of-fterms-fterms-of-hterms[simp]*: $\text{hterms-of-fterms } (\text{fterms-of-hterms } \text{ts}) = \text{ts}$

unfolding *hterms-of-fterms-def fterms-of-hterms-def* **by** (*simp add: map-idI*)

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

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

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

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

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

1.3.2 Conversions - Literals and Herbrand Literals

fun $flit\text{-of}\text{-hlit} :: hterm\ literal \Rightarrow fterm\ literal$ **where**
 $flit\text{-of}\text{-hlit} (Pos\ p\ ts) = Pos\ p\ (fterms\text{-of}\text{-hterms } ts)$
 $| flit\text{-of}\text{-hlit} (Neg\ p\ ts) = Neg\ p\ (fterms\text{-of}\text{-hterms } ts)$

fun $hlit\text{-of}\text{-flit} :: fterm\ literal \Rightarrow hterm\ literal$ **where**
 $hlit\text{-of}\text{-flit} (Pos\ p\ ts) = Pos\ p\ (hterms\text{-of}\text{-fterms } ts)$
 $| hlit\text{-of}\text{-flit} (Neg\ p\ ts) = Neg\ p\ (hterms\text{-of}\text{-fterms } ts)$

lemma $ground\text{-flit}\text{-of}\text{-hlit}$: $ground_l (flit\text{-of}\text{-hlit } l)$
by ($induction\ l$) ($simp\ add: ground\text{-fterms}\text{-of}\text{-hterms}$) $+$

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

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

lemma $sign\text{-flit}\text{-of}\text{-hlit}$: $sign (flit\text{-of}\text{-hlit } l) = sign\ l$ **by** ($cases\ l$) $auto$

lemma $hlit\text{-of}\text{-flit}\text{-bij}$: $bij\text{-betw } hlit\text{-of}\text{-flit } \{l. ground_l l\}$ $UNIV$
unfolding $bij\text{-betw}\text{-def}$

proof

show $inj\text{-on } hlit\text{-of}\text{-flit } \{l. ground_l l\}$ **using** $inj\text{-on}\text{-inverse}I$ $flit\text{-of}\text{-hlit}\text{-hlit}\text{-of}\text{-flit}$
by ($metis (mono\text{-}tags, lifting) mem\text{-}Collect\text{-}eq$)

next

have $\forall l. \exists l'. ground_l l' \wedge l = hlit\text{-of}\text{-flit } l'$

using $ground\text{-flit}\text{-of}\text{-hlit}$ $hlit\text{-of}\text{-flit}\text{-flit}\text{-of}\text{-hlit}$ **by** $metis$
then show $hlit\text{-of}\text{-flit } \{l. ground_l l\} = UNIV$ **by** $auto$

qed

lemma *flit-of-hlit-bij*: *bij-betw flit-of-hlit UNIV {l. ground_l l}*
unfolding *bij-betw-def inj-on-def*
proof
show $\forall x \in UNIV. \forall y \in UNIV. flit\text{-of}\text{-hlit } x = flit\text{-of}\text{-hlit } y \longrightarrow x = y$
using *ground-flit-of-hlit hlit-of-flit-flit-of-hlit* **by** *metis*
next
have $\forall l. ground_l l \longrightarrow (l = flit\text{-of}\text{-hlit } (hlit\text{-of}\text{-flit } l))$ **using** *hlit-of-flit-flit-of-hlit*
by *auto*
then have $\{l. ground_l l\} \subseteq flit\text{-of}\text{-hlit } ' UNIV$ **by** *blast*
moreover
have $\forall l. ground_l (flit\text{-of}\text{-hlit } l)$ **using** *ground-flit-of-hlit* **by** *auto*
ultimately show *flit-of-hlit ' UNIV = {l. ground_l l}* **using** *hlit-of-flit-flit-of-hlit*
ground-flit-of-hlit **by** *auto*
qed

1.3.3 Conversions - Atoms and Herbrand Atoms

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

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

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

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

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

lemma *hatom-of-fatom-bij*: *bij-betw hatom-of-fatom ground-fatoms UNIV*
unfolding *bij-betw-def*

proof
show *inj-on hatom-of-fatom ground-fatoms* **using** *inj-on-inverseI fatom-of-hatom-hatom-of-fatom*
unfolding *ground-fatoms-def*
by (*metis (mono-tags, lifting) mem-Collect-eq*)
next
have $\forall a. \exists a'. ground_{ts} (snd a') \wedge a = hatom\text{-of}\text{-fatom } a'$
using *ground-fatom-of-hatom hatom-of-fatom-fatom-of-hatom* **by** *metis*
then show *hatom-of-fatom ' ground-fatoms = UNIV* **unfolding** *ground-fatoms-def*
by *blast*
qed

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

1.4 Enumerations

1.4.1 Enumerating Strings

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

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

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

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

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

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

1.4.2 Enumerating Herbrand Atoms

definition *nat-of-hatom*:: *hterm atom* \Rightarrow *nat* **where**
nat-of-hatom \equiv (*SOME* *f*. *bij f*)

definition *hatom-of-nat*:: *nat* \Rightarrow *hterm atom* **where**
hatom-of-nat \equiv *inv nat-of-hatom*

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

lemma *infinite-hatoms*: *infinite* (*UNIV* :: ('*t atom*) *set*)

proof –

let *?diago* = λn . (*string-of-nat* *n*,[])

let *?undiago* = λa . *nat-of-string* (*fst a*)

have $\forall n$. *?undiago* (*?diago n*) = *n* **using** *nat-of-string-string-of-nat* **by** *auto*

moreover

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

ultimately show *infinite* (*UNIV* :: ('*t atom*) *set*) **using** *infinity*[*of ?undiago*
?diago UNIV] **by** *simp*

qed

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

proof –

let *?S* = *UNIV* :: (('*t::countable*) *atom*) *set*

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

moreover

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

ultimately

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

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

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

by *simp*

qed

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

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

unfolding *hatom-of-nat-def*

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

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

unfolding *hatom-of-nat-def*

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

1.4.3 Enumerating Ground Atoms

definition *fatom-of-nat* :: *nat* \Rightarrow *fterm atom* **where**
fatom-of-nat = ($\lambda n.$ *fatom-of-hatom* (*hatom-of-nat* *n*))

definition *nat-of-fatom* :: *fterm atom* \Rightarrow *nat* **where**
nat-of-fatom = ($\lambda t.$ *nat-of-hatom* (*hatom-of-fatom* *t*))

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

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

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

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

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

end

2 Trees

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

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

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

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

2.1 Sizes

fun *treesize* :: *tree* \Rightarrow *nat* **where**
treesize *Leaf* = 0

| $tree\ size\ (Branching\ l\ r) = 1 + tree\ size\ l + tree\ size\ r$

lemma *tree\ size-Leaf*:
assumes $tree\ size\ T = 0$
shows $T = Leaf$
using *assms* **by** (*cases T*) *auto*

lemma *tree\ size-Branching*:
assumes $tree\ size\ T = Suc\ n$
shows $\exists l\ r. T = Branching\ l\ r$
using *assms* **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*: $path\ p\ Leaf \longleftrightarrow p = []$
by (*induction p*) *auto*

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

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

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

lemma *path-inv-Branching*:
 $path\ p\ (Branching\ l\ r) \longleftrightarrow (p = [] \vee (\exists a\ p'. p = a\#\ p' \wedge (a \longrightarrow path\ p'\ l) \wedge (\neg a \longrightarrow path\ p'\ r)))$ (**is** $?L \longleftrightarrow ?R$)

proof

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

next

assume $r: ?R$

then show $?L$

proof

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

next

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

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

by *auto*

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

qed

qed

lemma *path-prefix*:

assumes *path* (*ds1* @ *ds2*) *T*

shows *path ds1 T*

using *assms* **proof** (*induction ds1 arbitrary: T*)

case (*Cons a ds1*)

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

then obtain *l r* **where** *p-lr: T = Branching l r* **by** *auto*

show *?case*

proof (*cases a*)

assume *atrue: a*

then have *path ((ds1) @ ds2) l* **using** *p-lr Cons(2) path-inv-Branching* **by**

auto

then have *path ds1 l* **using** *Cons(1)* **by** *auto*

then show *path (a # ds1) T* **using** *p-lr atrue* **by** *auto*

next

assume *afalse: ¬a*

then have *path ((ds1) @ ds2) r* **using** *p-lr Cons(2) path-inv-Branching* **by**

auto

then have *path ds1 r* **using** *Cons(1)* **by** *auto*

then show *path (a # ds1) T* **using** *p-lr afalse* **by** *auto*

qed

next

case (*Nil*) **then show** *?case* **by** *auto*

qed

2.3 Branches

fun *branch* :: *dir list* \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*: $\exists b. \text{branch } b T$

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:*

$\text{branch } (\text{Left}\#b) (\text{Branching } l r) \longleftrightarrow \text{branch } b l$

by *auto*

lemma *branch-inv-Branching-Right:*

$\text{branch } (\text{Right}\#b) (\text{Branching } l r) \longleftrightarrow \text{branch } b r$

by *auto*

lemma *branch-inv-Branching:*

$\text{branch } b (\text{Branching } l r) \longleftrightarrow$

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

by (*induction b*) *auto*

lemma *branch-inv-Leaf2:*

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

proof –

{

assume $T = \text{Leaf}$

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

}

moreover

{

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

then have $\forall b. \text{branch } b T \longrightarrow \neg(\exists a b'. b = a \# b')$ **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:*

assumes *branch ds T*

shows *path ds T*

using *assms* **proof** (*induction T arbitrary: ds*)

case *Leaf*

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

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

next

case (*Branching 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:*

assumes $T = \text{Branching } T_1 T_2$

shows $(\exists B. \text{branch } (B@[True]) T \wedge \text{branch } (B@[False]) T)$

```

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

```

2.4 Internal Paths

```

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

```

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

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

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

lemma *internal-inv-Branching*:

internal p (Branching l r) \longleftrightarrow (p= \square) \vee (\exists a p'. p=a#p' \wedge (a \longrightarrow internal p' l) \wedge (\neg a \longrightarrow internal p' r))) (is ?L \longleftrightarrow ?R)

proof

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

next

assume r: ?R

then show ?L

proof

assume p = \square then show ?L by auto

next

assume \exists a p'. p=a#p' \wedge (a \longrightarrow internal p' l) \wedge (\neg a \longrightarrow internal p' r)

then obtain a p' where p=a#p' \wedge (a \longrightarrow internal p' l) \wedge (\neg a \longrightarrow internal p' r) by auto

then show ?L by (cases a) auto

qed

qed

lemma *internal-is-path*:

assumes internal ds T

shows path ds T

using assms **proof** (induction T arbitrary: ds)

case Leaf

then have False using internal-inv-Leaf by auto

then show ?case by auto

next

case (Branching T₁ T₂)

then obtain a b where ds-p: ds= \square \vee ds = a # b \wedge (a \longrightarrow internal b T₁) \wedge (\neg a \longrightarrow internal b T₂) using internal-inv-Branching by blast

then have ds = \square \vee (a \longrightarrow path b T₁) \wedge (\neg a \longrightarrow path b T₂) using Branching by auto

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

qed

lemma *internal-prefix*:

assumes internal (ds1@ds2@[d]) T

shows internal ds1 T

using assms **proof** (induction ds1 arbitrary: T)

case (Cons a ds1)

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

then obtain l r where p-lr: T = Branching l r by auto

show ?case

proof (cases a)

assume atrue: a

then have internal ((ds1) @ ds2 @ [d]) l using p-lr Cons(2) internal-inv-Branching by auto

then have internal ds1 l using Cons(1) by auto

then show internal (a # ds1) T using p-lr atrue by auto

next

```

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

```

```

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

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

lemma *path-delete*:

assumes *path* *p* (*delete* *ds* *T*)

shows *path* *p* *T*

using *assms* **proof** (*induction* *p* *arbitrary*: *T* *ds*)

case *Nil*

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

next

case (*Cons* *a* *p*)

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

have \exists *dT1* *dT2*. *delete* *ds* *T* = *Branching* *dT1* *dT2* **using** *Cons* *path-inv-Cons*
by *auto*

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

then have \exists *T1* *T2*. *T*=*Branching* *T1* *T2*

by (*cases* *T*; *cases* *ds*) *auto*

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

{

assume *a-p*: *a*

assume *b-p*: \neg *b*

have *path* (*a* # *p*) (*delete* *ds* *T*) **using** *Cons* **by** $-$

then have *path* (*a* # *p*) (*Branching* (*T1*) (*delete* *ds'* *T2*)) **using** *b-p* *bds'-p*

T1T2-p **by** *auto*

then have *path* *p* *T1* **using** *a-p* **by** *auto*

then have *?case* **using** *T1T2-p* *a-p* **by** *auto*

}

moreover

{

assume *a-p*: \neg *a*

assume *b-p*: *b*

have *path* (*a* # *p*) (*delete* *ds* *T*) **using** *Cons* **by** $-$

then have *path* (*a* # *p*) (*Branching* (*delete* *ds'* *T1*) *T2*) **using** *b-p* *bds'-p*

T1T2-p **by** *auto*

then have *path* *p* *T2* **using** *a-p* **by** *auto*

then have *?case* **using** *T1T2-p* *a-p* **by** *auto*

}

moreover

{

assume *a-p*: *a*

assume *b-p*: *b*

have *path* (*a* # *p*) (*delete* *ds* *T*) **using** *Cons* **by** $-$

then have *path* (*a* # *p*) (*Branching* (*delete* *ds'* *T1*) *T2*) **using** *b-p* *bds'-p*

T1T2-p **by** *auto*


```

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

```

lemma *branch-delete*:

```

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

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

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

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

```

```

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

```

lemma *branch-delete-postfix*:

```

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

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

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

```

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

```

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

```

lemma *treezise-delete*:

```

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

```

```

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

```

```

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

```

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

lemma *treesize-cutoff*: $treesize (cutoff red ds T) \leq treesize T$

proof (*induction T arbitrary: ds*)

case *Leaf* then show ?case by auto

next

case (*Branching T1 T2*)

then have $treesize (cutoff red (ds@[Left]) T1) + treesize (cutoff red (ds@[Right]) T2) \leq treesize T1 + treesize T2$ using *add-mono* by *blast*

then show ?case by auto

qed

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

anypath T P $\equiv \forall p. path p T \longrightarrow P p$

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

anybranch T P $\equiv \forall p. branch p T \longrightarrow P p$

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

lemma *cutoff-branch'*:

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

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

```

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

```

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

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

from Branching have ∀ p. branch (Right#p) (Branching T₁ T₂) → red (ds @ (Right#p)) by blast
 then have ∀ p. branch p T₂ → red (ds @ (Right#p)) by auto
 then have anybranch T₂ (λp. red ((ds @ [Right]) @ p)) by auto
 then have bb: anyinternal (cutoff red (ds @ [Right]) T₂) (λp. ¬ red ((ds @ [Right]) @ p)) using Branching by blast

```

{
  fix p
  assume b-p: internal p ?T
  then have ds-p: ¬red ds using internal-inv-Leaf by auto
  have p=[] ∨ p≠[] by auto
  then have ¬red(ds@p)
  proof
    assume p=[] then show ¬red(ds@p) using ds-p by auto
  next
    let ?T1' = cutoff red (ds@[Left]) T1
    let ?T2' = cutoff red (ds@[Right]) T2
    assume p≠[]
    moreover
    have ?T = Branching ?T1' ?T2' using ds-p by auto
  qed
}

```

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

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

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

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

3 Possibly Infinite Trees

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

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

The subtree in with root r

fun *subtree* $:: \text{dir list set} \Rightarrow \text{dir list} \Rightarrow \text{dir list set}$ **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*:
 $subtree\ T\ ds \subseteq subtree\ T\ (ds @ [\text{Left}]) \cup subtree\ T\ (ds @ [\text{Right}]) \cup \{ds\}$
proof (*rule subsetI; rule Set.UnCI*)
let $?subtree = subtree\ T$
fix x
assume $asm: x \in ?subtree\ ds$
assume $x \notin \{ds\}$

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

3.1 Infinite Paths

abbreviation *wf-infpath* :: $(nat \Rightarrow 'a list) \Rightarrow bool$ **where**
 $wf_infpath\ f \equiv (f\ 0 = []) \wedge (\forall n. \exists a. f\ (Suc\ n) = (f\ n) @ [a])$

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

lemma *chain-prefix*:
assumes *wf-infpath f*
assumes $n_1 \leq n_2$
shows $\exists a. (f\ n_1) @ a = (f\ n_2)$
using *assms* **proof** (*induction n₂*)
case (*Suc n₂*)
then have $n_1 \leq n_2 \vee n_1 = Suc\ n_2$ **by** *auto*
then show *?case*
proof
assume $n_1 \leq n_2$
then obtain *a* **where** $f\ n_1 @ a = f\ n_2$ **using** *Suc* **by** *auto*
have $b: \exists b. f\ (Suc\ n_2) = f\ n_2 @ [b]$ **using** *Suc* **by** *auto*
from *a b* **have** $\exists b. f\ n_1 @ (a @ [b]) = f\ (Suc\ n_2)$ **by** *auto*
then show $\exists c. f\ n_1 @ c = f\ (Suc\ n_2)$ **by** *blast*
next
assume $n_1 = Suc\ n_2$
then have $f\ n_1 @ [] = f\ (Suc\ n_2)$ **by** *auto*
then show $\exists a. f\ n_1 @ a = f\ (Suc\ n_2)$ **by** *auto*
qed
qed *auto*

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

lemma *ith-in-extension*:
assumes *chain: wf-infpath f*
assumes *smalli: i < length (f n₁)*
assumes $n_1 n_2: n_1 \leq n_2$

shows $f\ n_1\ !\ i = f\ n_2\ !\ i$
proof –
from *chain* $n_1 n_2$ **have** $\exists a. f\ n_1\ @\ a = f\ n_2$ **using** *chain-prefix* **by** *blast*
then obtain a **where** $a\text{-}p: f\ n_1\ @\ a = f\ n_2$ **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*: $\forall ds\ d. (ds\ @\ d) \in T \longrightarrow ds \in T$ **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 by blast
    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 by blast

  then have  $(?c\ (Suc\ n)) \in T$  by auto
  then show ?case using next-next-inf by auto
qed
}
then show wf-infpth  $?c \wedge (\forall n. (?c\ n) \in T)$  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 l₁*] **by** *auto*

then show *?thesis* **using** *cancel-comp1*[*of l₂*] **by** *auto*

qed

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

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

proof

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

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

proof –

have $l_1^c = l_2 \longrightarrow l_1 = l_2^c$ **using** *cancel-comp1*[*of l₁*] **by** *auto*

moreover

have $l_1 = l_2^c \longrightarrow l_1^c = l_2$ **using** *cancel-comp1* **by** *auto*

ultimately

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

qed

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

lemma *sign-comp-atom*: $\text{sign } l_1 \neq \text{sign } l_2 \wedge \text{get-atom } l_1 = \text{get-atom } l_2 \longleftrightarrow l_2 = l_1^c$
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**
 $L^C \equiv \text{complement } 'L$

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

lemma *cancel-compls2*:

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**

$\text{vars}_t (\text{Var } x) = \{x\}$

$|\text{vars}_t (\text{Fun } f \text{ ts}) = (\bigcup t \in \text{set } \text{ts}. \text{vars}_t t)$

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

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

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

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

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

definition $eval_{cs} :: 'u \text{ fun-denot} \Rightarrow 'u \text{ pred-denot} \Rightarrow \text{fterm clause set} \Rightarrow \text{bool}$
where
 $eval_{cs} F G Cs \longleftrightarrow (\forall C \in Cs. eval_c F G C)$

7.1 Semantics of Ground Terms

lemma *ground-var-denott*:
assumes $ground_t t$
shows $eval_t E F t = eval_t E' F t$
using *assms* **proof** (*induction t*)
case ($Var x$)
then have *False* **by** *auto*
then show *?case* **by** *auto*
next
case ($Fun f ts$)
then have $\forall t \in \text{set } ts. ground_t t$ **by** *auto*
then have $\forall t \in \text{set } ts. eval_t E F t = eval_t E' F t$ **using** *Fun* **by** *auto*
then have $eval_{ts} E F ts = eval_{ts} E' F ts$ **by** *auto*
then have $F f (\text{map } (eval_t E F) ts) = F f (\text{map } (eval_t E' F) ts)$ **by** *metis*
then show *?case* **by** *simp*
qed

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

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

8 Substitutions

type-synonym $substitution = \text{var-sym} \Rightarrow \text{fterm}$

fun $sub :: \text{fterm} \Rightarrow substitution \Rightarrow \text{fterm}$ (**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 $\varepsilon :: substitution$ **where**
 $\varepsilon \equiv Var$

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Substitutions and Ground Terms

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

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

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

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

lemma *ground_{l_s}-subs*:

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

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

proof –

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

moreover

{
 fix *l*
 assume *l-L*: $l \in L \cdot_{l_s} \sigma$
 then obtain *l'-p*: $l' \in L \wedge l' \cdot_l \sigma = l$ **by** *auto*
 then have $l' = l$ **using** *ground* *ground_l-subs* **by** *auto*
 from *l-L l'-p* **this have** $l \in L$ **by** *auto*
}

ultimately show *?thesis* **by** *auto*

qed

8.3 Composition

definition *composition* :: *substitution* \Rightarrow *substitution* \Rightarrow *substitution* (**infixl** · 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-subt by auto
qed

```

```

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

```

```

lemma instance-oft-trans :
  assumes  $t_{12}$ : instance-oft  $t_1$   $t_2$ 
  assumes  $t_{23}$ : instance-oft  $t_2$   $t_3$ 
  shows instance-oft  $t_1$   $t_3$ 
proof –
  from  $t_{12}$  obtain  $\sigma_{12}$  where  $t_1 = t_2 \cdot_t \sigma_{12}$ 
    unfolding instance-oft-def by auto
  moreover
  from  $t_{23}$  obtain  $\sigma_{23}$  where  $t_2 = t_3 \cdot_t \sigma_{23}$ 
    unfolding instance-oft-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-oft-def by auto
qed

```

```

lemma instance-ofts-trans :
  assumes  $ts_{12}$ : instance-ofts  $ts_1$   $ts_2$ 
  assumes  $ts_{23}$ : instance-ofts  $ts_2$   $ts_3$ 
  shows instance-ofts  $ts_1$   $ts_3$ 
proof –
  from  $ts_{12}$  obtain  $\sigma_{12}$  where  $ts_1 = ts_2 \cdot_{ts} \sigma_{12}$ 
    unfolding instance-ofts-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-conseq2l_s* **by** *simp*
then show *?thesis unfolding instance-of_{l_s}-def* **by** *auto*
qed

8.4 Merging substitutions

lemma *project-sub*:
assumes *inst-C*: $C \cdot_{l_s} lmbd = C'$
assumes *L'sub*: $L' \subseteq C'$
shows $\exists L \subseteq C. L \cdot_{l_s} lmbd = L' \wedge (C - L) \cdot_{l_s} lmbd = C' - L'$
proof –
let $?L = \{l \in C. \exists l' \in L'. l \cdot_l lmbd = l'\}$

have $?L \subseteq C$ **by** *auto*
moreover
have $?L \cdot_{l_s} \text{ lmbd} = L'$
proof (*rule Orderings.order-antisym; rule Set.subsetI*)
fix l'
assume $l': l' \in L'$
from *inst-C* **have** $\{l \cdot_l \text{ lmbd} \mid l. l \in C\} = C'$ **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** $L'_{\text{sub}} l'L$ **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*:
assumes $\forall x \in \text{vars}_t t. \sigma_1 x = \sigma_2 x$
shows $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$
using *assms* **proof** (*induction t*)
case (*Fun f ts*)
have $f: \forall t. t \in \text{set } ts \longrightarrow \text{vars}_t t \subseteq \text{vars}_{ts} ts$ **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** $f \text{ Fun}(2)$ **by** *auto*
then show $t \cdot_t \sigma_1 = t \cdot_t \sigma_2$ **using** *Fun tints* **by** *auto*
qed
then have $ts \cdot_{ts} \sigma_1 = ts \cdot_{ts} \sigma_2$ **by** *auto*
then show *?case* **by** *auto*
qed *auto*

lemma *relevant-vars-subts*:
assumes *asm*: $\forall x \in \text{vars}_{ts} ts. \sigma_1 x = \sigma_2 x$
shows $ts \cdot_{ts} \sigma_1 = ts \cdot_{ts} \sigma_2$
proof $-$
have $f: \forall t. t \in \text{set } ts \longrightarrow \text{vars}_t t \subseteq \text{vars}_{ts} ts$ **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** $f \text{ asm}$ **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*:
assumes $\forall x \in \text{vars}_l l. \sigma_1 x = \sigma_2 x$
shows $l \cdot_l \sigma_1 = l \cdot_l \sigma_2$
using *assms* **proof** (*induction l*)
case (*Pos p ts*)
then show *?case* **using** *relevant-vars-subts unfolding vars_l-def* **by** *auto*
next
case (*Neg p ts*)
then show *?case* **using** *relevant-vars-subts unfolding vars_l-def* **by** *auto*
qed

lemma *relevant-vars-subls*:
assumes *asm*: $\forall x \in \text{vars}_{l_s} L. \sigma_1 x = \sigma_2 x$
shows $L \cdot_{l_s} \sigma_1 = L \cdot_{l_s} \sigma_2$
proof –
have *f*: $\forall l. l \in L \longrightarrow \text{vars}_l l \subseteq \text{vars}_{l_s} L$ **unfolding** *vars_{l_s}-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 *?η* = $\lambda x. \text{if } x \in \text{vars}_{l_s} C \text{ then } \text{lmbd } x \text{ else } \mu x$
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 ?η 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 ?η μ]* **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

$std_1 C \equiv C \cdot_{ls} (\lambda x. Var ("1" @ x))$

abbreviation $std_2 :: fterm\ clause \Rightarrow fterm\ clause$ **where**
 $std_2 C \equiv C \cdot_{ls} (\lambda x. Var ("2" @ x))$

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

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

lemma *std-apart-apart*: $vars_{ls} (std_1 C_1) \cap vars_{ls} (std_2 C_2) = \{\}$

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

lemma *std-apart-instance-of_ls1*: $instance-of_{ls} C_1 (std_1 C_1)$

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

have $C_1 \cdot_{ls} \varepsilon = C_1$ **using** *empty-subls* **by** *auto*
then **have** $C_1 \cdot_{ls} ((\lambda x. Var ("1" @ x)) \cdot (\lambda x. Var (tl x))) = C_1$ **using** *empty* **by**
auto
then **have** $(C_1 \cdot_{ls} (\lambda x. Var ("1" @ x))) \cdot_{ls} (\lambda x. Var (tl x)) = C_1$ **using** *composition-conseq2ls*
by *auto*

then **have** $C_1 = (std_1 C_1) \cdot_{ls} (\lambda x. Var (tl x))$ **by** *auto*
then **show** *instance-of_ls C_1 (std_1 C_1)* **unfolding** *instance-of_ls-def* **by** *auto*
qed

lemma *std-apart-instance-of_ls2*: $instance-of_{ls} C_2 (std_2 C_2)$

proof –

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

have $C2 \cdot_{ls} \varepsilon = C2$ **using** *empty-subls* **by** *auto*
then have $C2 \cdot_{ls} ((\lambda x. \text{Var } ("2"@x)) \cdot (\lambda x. \text{Var } (tl\ x))) = C2$ **using** *empty*
by *auto*
then have $(C2 \cdot_{ls} (\lambda x. \text{Var } ("2"@x))) \cdot_{ls} (\lambda x. \text{Var } (tl\ x)) = C2$ **using** *composition-conseq2ls*
by *auto*
then have $C2 = (std_2\ C2) \cdot_{ls} (\lambda x. \text{Var } (tl\ x))$ **by** *auto*
then show *instance-of_{ls} C2 (std₂ C2)* **unfolding** *instance-of_{ls}-def* **by** *auto*
qed

9 Unifiers

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

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

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

unifier_{ls} $\sigma\ L \longleftrightarrow (\exists l'. \forall l \in L. l \cdot_l \sigma = l')$

lemma *unif-sub*:

assumes *unif*: *unifier_{ls}* $\sigma\ L$

assumes *nonempty*: $L \neq \{\}$

shows $\exists l. \text{subls } L\ \sigma = \{\text{subl } l\ \sigma\}$

proof –

from *nonempty* **obtain** l **where** $l \in L$ **by** *auto*

from *unif* **this have** $L \cdot_{ls} \sigma = \{l \cdot_l \sigma\}$ **unfolding** *unifier_{ls}-def* **by** *auto*

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

qed

lemma *unifiert-def2*:

assumes *L-elem*: $ts \neq \{\}$

shows *unifier_{ts}* $\sigma\ ts \longleftrightarrow (\exists l. (\lambda t. \text{sub } t\ \sigma) \text{ ' } ts = \{l\})$

proof

assume *unif*: *unifier_{ts}* $\sigma\ ts$

from *L-elem* **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 $\forall l' \in ts. l' \cdot_{ts} \sigma = l$ **by** *auto*

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

qed

lemma *unifier_{ls}-def2*:

assumes *L-elem*: $L \neq \{\}$

shows *unifier_{ls}* $\sigma\ L \longleftrightarrow (\exists l. L \cdot_{ls} \sigma = \{l\})$

proof

assume *unif*: $\text{unifier}_{l_s} \sigma L$

from *L*-elem 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}*: $\text{ground}_{l_s} L$

assumes *unif*: $\text{unifier}_{l_s} \sigma' L$

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 $\longleftrightarrow (\exists \sigma. \text{unifier}_{t_s} \sigma fs)$

definition *unifiablels* :: *fterm literal set* \Rightarrow *bool* **where**

unifiablels L $\longleftrightarrow (\exists \sigma. \text{unifier}_{l_s} \sigma L)$

lemma *unifier-comp[simp]*: $\text{unifier}_{l_s} \sigma (L^C) \longleftrightarrow \text{unifier}_{l_s} \sigma L$

proof

assume $\text{unifier}_{l_s} \sigma (L^C)$

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

unfolding *unifier_{l_s}-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_{l_s} σ L unfolding unifier_{l_s}-def* by *auto*

next

assume $\text{unifier}_{l_s} \sigma L$

then obtain *l'* where *l'-p*: $\forall l \in L. l \cdot_l \sigma = l'$ unfolding *unifier_{l_s}-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 $l \cdot_l \sigma = (l^c)^c$ **using** *l'-p comp-sub cancel-comp1* **by** *metis*
qed
then show $\text{unifier}_{l_s} \sigma (L^C)$ **unfolding** *unifier_{l_s}-def* **by** *auto*
qed

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

lemma *unifier-sub2*:
assumes *asm*: $\text{unifier}_{l_s} \sigma (L_1 \cup L_2)$
shows $\text{unifier}_{l_s} \sigma L_1 \wedge \text{unifier}_{l_s} \sigma L_2$

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 $\text{mgu}_{t_s} :: \text{substitution} \Rightarrow \text{fterm set} \Rightarrow \text{bool}$ **where**
 $\text{mgu}_{t_s} \sigma ts \longleftrightarrow \text{unifier}_{t_s} \sigma ts \wedge (\forall u. \text{unifier}_{t_s} u ts \longrightarrow (\exists i. u = \sigma \cdot i))$

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

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**

$\text{applicable } C_1 C_2 L_1 L_2 \sigma \longleftrightarrow$
 $C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
 $\wedge \text{vars}_{l_s} C_1 \cap \text{vars}_{l_s} C_2 = \{\}$
 $\wedge L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
 $\wedge \text{mgu}_{l_s} \sigma (L_1 \cup L_2^C)$

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**
 $\text{mresolution } C_1 C_2 L_1 L_2 \sigma = ((C_1 \cdot_{l_s} \sigma) - (L_1 \cdot_{l_s} \sigma)) \cup ((C_2 \cdot_{l_s} \sigma) - (L_2 \cdot_{l_s} \sigma))$

definition $\text{resolution} :: \text{fterm clause} \Rightarrow \text{fterm clause}$

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

inductive *mresolution-step* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**
mresolution-rule:

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

| *standardize-apart*:

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

inductive *resolution-step* :: *fterm clause set* \Rightarrow *fterm clause set* \Rightarrow *bool* **where**
resolution-rule:

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

| *standardize-apart*:

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

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

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

11 Soundness

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

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

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

apply (*induction t*)

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

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

done

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

using *substitution* **by** *auto*

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

apply (*induction l*)

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

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

done

lemma *subst-sound*:

assumes *asm*: $eval_c F G C$

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

unfolding *eval_c-def* **proof**

fix E
from asm **have** $\forall E'. \exists l \in C. eval_l E' F G l$ **using** $eval_c-def$ **by** $blast$
then have $\exists l \in C. eval_l (evalsub E F \sigma) F G l$ **by** $auto$
then show $\exists l \in C \cdot_l \sigma. eval_l E F G l$ **using** $substitution$ **by** $blast$
qed

lemma $simple-resolution-sound$:

assumes $C_1 sat$: $eval_c F G C_1$
assumes $C_2 sat$: $eval_c F G C_2$
assumes $l_1 inc_1$: $l_1 \in C_1$
assumes $l_2 inc_2$: $l_2 \in C_2$
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** $(cases l_1) auto$
then show $\exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\})))$. $eval_l E F G l$
proof
assume $eval_l E F G l_1$
then have $\neg eval_l E F G l_2$ **using** $comp$ **by** $(cases l_1) auto$
then have $\exists l_2' \in C_2. l_2' \neq l_2 \wedge eval_l E F G l_2'$ **using** $l_2 inc_2 C_2 sat$
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** $sat_1 \sigma$: $eval_c F G (C_1 \cdot_l \sigma)$ **using** $subst-sound$ **by** $blast$
from sat_2 **have** $sat_2 \sigma$: $eval_c F G (C_2 \cdot_l \sigma)$ **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 $inc_1 \sigma$: $l_1 \cdot_l \sigma \in C_1 \cdot_l \sigma$ **by** $auto$

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

from l_1 - p *appl* **have** $l_1\sigma isl_1\sigma: \{l_1 \cdot_l \sigma\} = L_1 \cdot_{l_s} \sigma$
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** $unifier_{l_s} \sigma L_2$ **by** *auto*
from *this* l_2 - p **have** $l_2\sigma isl_2\sigma: \{l_2 \cdot_l \sigma\} = L_2 \cdot_{l_s} \sigma$ **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 C₁ ·_{l_s} σ C₂ ·_{l_s} σ l₁ ·_l σ l₂ ·_l σ]*
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 C₁ C₂ L₁ L₂ σ ⊆ resolution C₁ C₂ L₁ L₂ σ*
unfolding *mresolution-def resolution-def* **by** *auto*

lemma *superset-sound:*
assumes *sup: C ⊆ C'*
assumes *sat: eval_c F G C*
shows $eval_c F G C'$
proof –
have $\forall E. \exists l \in C'. eval_l E F G l$
proof
fix E
from *sat* **have** $\forall E. \exists l \in C. eval_l E F G l$ **unfolding** *eval_c-def* **by** –
then **have** $\exists l \in C. eval_l E F G l$ **by** *auto*
then **show** $\exists l \in C'. eval_l E F G l$ **using** *sup* **by** *auto*
qed
then **show** $eval_c F G C'$ **unfolding** *eval_c-def* **by** *auto*
qed

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

lemma *mstep-sound*:
assumes *mresolution-step* $Cs Cs'$
assumes $eval_{cs} F G Cs$
shows $eval_{cs} F G Cs'$
using *assms* **proof** (*induction rule: mresolution-step.induct*)
case (*mresolution-rule* $C_1 Cs C_2 l_1 l_2 \sigma$)
then have $eval_c F G C_1 \wedge eval_c F G C_2$ **unfolding** *eval_{cs}-def* **by** *auto*
then have $eval_c F G$ (*mresolution* $C_1 C_2 l_1 l_2 \sigma$)
using *mresolution-sound mresolution-rule* **by** *auto*
then show *?case* **using** *mresolution-rule unfolding eval_{cs}-def* **by** *auto*
next
case (*standardize-apart* $C Cs C'$)
then have $eval_c F G C$ **unfolding** *eval_{cs}-def* **by** *auto*
then have $eval_c F G C'$ **using** *subst-sound standardize-apart unfolding var-renaming-of-def instance-of_{1s}-def* **by** *metis*
then show *?case* **using** *standardize-apart unfolding eval_{cs}-def* **by** *auto*
qed

theorem *step-sound*:
assumes *resolution-step* $Cs Cs'$
assumes $eval_{cs} F G Cs$
shows $eval_{cs} F G Cs'$
using *assms* **proof** (*induction rule: resolution-step.induct*)
case (*resolution-rule* $C_1 Cs C_2 l_1 l_2 \sigma$)
then have $eval_c F G C_1 \wedge eval_c F G C_2$ **unfolding** *eval_{cs}-def* **by** *auto*
then have $eval_c F G$ (*resolution* $C_1 C_2 l_1 l_2 \sigma$)
using *resolution-sound resolution-rule* **by** *auto*
then show *?case* **using** *resolution-rule unfolding eval_{cs}-def* **by** *auto*
next
case (*standardize-apart* $C Cs C'$)
then have $eval_c F G C$ **unfolding** *eval_{cs}-def* **by** *auto*
then have $eval_c F G C'$ **using** *subst-sound standardize-apart unfolding var-renaming-of-def instance-of_{1s}-def* **by** *metis*
then show *?case* **using** *standardize-apart unfolding eval_{cs}-def* **by** *auto*
qed

lemma *mderivation-sound*:
assumes *mresolution-deriv* $Cs Cs'$

assumes $eval_{cs} F G Cs$
shows $eval_{cs} F G Cs'$
using *assms unfolding mresolution-deriv-def*
proof (*induction rule: rtrancl.induct*)
case *rtrancl-refl* **then show** *?case* **by** *auto*
next
case (*rtrancl-into-rtrancl Cs₁ Cs₂ Cs₃*) **then show** *?case* **using** *mstep-sound*
by *auto*
qed

theorem *derivation-sound*:
assumes *resolution-deriv Cs Cs'*
assumes $eval_{cs} F G Cs$
shows $eval_{cs} F G Cs'$
using *assms unfolding resolution-deriv-def*
proof (*induction rule: rtrancl.induct*)
case *rtrancl-refl* **then show** *?case* **by** *auto*
next
case (*rtrancl-into-rtrancl Cs₁ Cs₂ Cs₃*) **then show** *?case* **using** *step-sound* **by**
auto
qed

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

12 Herbrand Interpretations

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

term *HFun*

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

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

lemma *eval_l-ground_{ts}*:
assumes *asm: ground_{ts} ts*
shows $\text{eval}_l E \text{HFun } G (\text{Pos } P \text{ } ts) \longleftrightarrow G P (\text{hterms-of-ftersms } ts)$
proof –
have $\text{eval}_l E \text{HFun } G (\text{Pos } P \text{ } ts) = G P (\text{eval}_{ts} E \text{HFun } ts)$ **by** *auto*
also have $\dots = G P (\text{hterms-of-ftersms } ts)$ **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**
 $\text{falsifies}_l G l \longleftrightarrow$
 $\text{ground}_l l$
 $\wedge (\text{let } i = \text{nat-of-fatom } (\text{get-atom } l) \text{ in}$
 $i < \text{length } G \wedge G ! i = (\neg \text{sign } l)$
 $)$

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

abbreviation *falsifies_g* :: *partial-pred-denot* \Rightarrow *fterm clause* \Rightarrow *bool* **where**
 $\text{falsifies}_g G C \equiv \text{ground}_{ts} C \wedge (\forall l \in C. \text{falsifies}_l G l)$

abbreviation *falsifies_c* :: *partial-pred-denot* \Rightarrow *fterm clause* \Rightarrow *bool* **where**
 $\text{falsifies}_c G C \equiv (\exists C'. \text{instance-of}_{ts} C' C \wedge \text{falsifies}_g G C')$

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

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

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

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

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

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

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

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

proof –
have $ground_t (\text{sub-of-denot } E x)$ **using** *ground-fterm-of-hterm* **by** *simp*
then
have $eval_t E \text{ HFun } (\text{sub-of-denot } E x) = \text{hterm-of-fterm } (\text{sub-of-denot } E x)$
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*:

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

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

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

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

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

show *?case* **by** *blast*
qed

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

lemma *sub-of-denot-equiv-ground'*:

$eval_l E \text{ HFun } G l = eval_l E \text{ HFun } G (l \cdot_l \text{ sub-of-denot } E) \wedge ground_l (l \cdot_l \text{ sub-of-denot } E)$

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 *falsifies_c* $G (C \cdot_{l_s} \tau)$

shows *falsifies_c* $G C$

proof –

from *assms* **obtain** C' **where** $C'-p$: *instance-of_{l_s}* $C' (C \cdot_{l_s} \tau) \wedge \text{falsifies}_g G C'$ **by** *auto*

then have *instance-of_{l_s}* $(C \cdot_{l_s} \tau) C$ **unfolding** *instance-of_{l_s}-def* **by** *auto*

then have *instance-of_{l_s}* $C' C$ **using** $C'-p$ *instance-of_{l_s}-trans* **by** *auto*

then show *?thesis* **using** $C'-p$ **by** *auto*

qed

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

lemma *sub-of-denot-equiv-ground*:

$((\exists l \in C. eval_l E \text{ HFun } G l) \longleftrightarrow (\exists l \in C \cdot_{l_s} \text{ sub-of-denot } E. eval_l E \text{ HFun } G l))$

$\wedge ground_{l_s} (C \cdot_{l_s} \text{ sub-of-denot } E)$

using *sub-of-denot-equiv-ground'* **by** *auto*

lemma *std₁-falsifies*: *falsifies_c* $G C_1 \longleftrightarrow \text{falsifies}_c G (\text{std}_1 C_1)$

proof

assume *asm*: *falsifies_c* $G C_1$

then obtain Cg **where** *instance-of_{l_s}* $Cg C_1 \wedge \text{falsifies}_g G Cg$ **by** *auto*

moreover

then have *instance-of_{l_s}* $Cg (\text{std}_1 C_1)$ **using** *std-apart-instance-of_{l_s}1* *instance-of_{l_s}-trans* **by** *blast*

ultimately

show *falsifies_c* $G (\text{std}_1 C_1)$ **by** *auto*

next

assume *asm*: *falsifies_c* $G (\text{std}_1 C_1)$

then have *inst*: *instance-of_{l_s}* $(\text{std}_1 C_1) C_1$ **unfolding** *instance-of_{l_s}-def* **by** *auto*

from *asm* **obtain** Cg **where** *instance-of_{l_s}* $Cg (\text{std}_1 C_1) \wedge \text{falsifies}_g G Cg$ **by** *auto*

moreover

then have $instance-of_{1s} Cg C_1$ **using** $inst\ instance-of_{1s}-trans$ **by** $blast$
ultimately
show $falsifies_c G C_1$ **by** $auto$
qed

lemma $std_2-falsifies: falsifies_c G C_2 \longleftrightarrow falsifies_c G (std_2 C_2)$

proof

assume $asm: falsifies_c G C_2$
then obtain Cg **where** $instance-of_{1s} Cg C_2 \wedge falsifies_g G Cg$ **by** $auto$
moreover
then have $instance-of_{1s} Cg (std_2 C_2)$ **using** $std-apart-instance-of_{1s}2\ instance-of_{1s}-trans$
by $blast$
ultimately
show $falsifies_c G (std_2 C_2)$ **by** $auto$
next
assume $asm: falsifies_c G (std_2 C_2)$
then have $inst: instance-of_{1s} (std_2 C_2) C_2$ **unfolding** $instance-of_{1s}-def$ **by**
 $auto$

from asm **obtain** Cg **where** $instance-of_{1s} Cg (std_2 C_2) \wedge falsifies_g G Cg$ **by**
 $auto$
moreover
then have $instance-of_{1s} Cg C_2$ **using** $inst\ instance-of_{1s}-trans$ **by** $blast$
ultimately
show $falsifies_c G C_2$ **by** $auto$
qed

lemma $std_1-renames: var-renaming-of C_1 (std_1 C_1)$

proof –

have $instance-of_{1s} C_1 (std_1 C_1)$ **using** $std-apart-instance-of_{1s}1$ **by** $auto$
moreover have $instance-of_{1s} (std_1 C_1) C_1$ **unfolding** $instance-of_{1s}-def$ **by**
 $auto$
ultimately show $var-renaming-of C_1 (std_1 C_1)$ **unfolding** $var-renaming-of-def$
by $auto$
qed

lemma $std_2-renames: var-renaming-of C_2 (std_2 C_2)$

proof –

have $instance-of_{1s} C_2 (std_2 C_2)$ **using** $std-apart-instance-of_{1s}2$ **by** $auto$
moreover have $instance-of_{1s} (std_2 C_2) C_2$ **unfolding** $instance-of_{1s}-def$ **by**
 $auto$
ultimately show $var-renaming-of C_2 (std_2 C_2)$ **unfolding** $var-renaming-of-def$
by $auto$
qed

14 Semantic Trees

abbreviation $closed-branch :: partial-pred-denot \Rightarrow tree \Rightarrow fterm\ clause\ set \Rightarrow bool$ **where**

$closed\text{-}branch\ G\ T\ Cs \equiv branch\ G\ T \wedge falsifies_{cs}\ G\ Cs$

abbreviation(*input*) $open\text{-}branch :: partial\text{-}pred\text{-}denot \Rightarrow tree \Rightarrow fterm\ clause\ set$
 $\Rightarrow bool$ **where**
 $open\text{-}branch\ G\ T\ Cs \equiv branch\ G\ T \wedge \neg falsifies_{cs}\ G\ Cs$

definition $closed\text{-}tree :: tree \Rightarrow fterm\ clause\ set \Rightarrow bool$ **where**
 $closed\text{-}tree\ T\ Cs \longleftrightarrow anybranch\ T\ (\lambda b. closed\text{-}branch\ b\ T\ Cs)$
 $\wedge anyinternal\ T\ (\lambda p. \neg falsifies_{cs}\ p\ Cs)$

15 Herbrand's Theorem

lemma *maximum*:

assumes $asm: finite\ C$
shows $\exists n :: nat. \forall l \in C. fl \leq n$

proof

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

lemma *extend-preserves-model*:

assumes $f\text{-infpath}: wf\text{-infpath}\ (f :: nat \Rightarrow partial\text{-}pred\text{-}denot)$
assumes $C\text{-ground}: ground_{I_s}\ C$
assumes $C\text{-sat}: \neg falsifies_c\ (f\ (Suc\ n))\ C$
assumes $n\text{-max}: \forall l \in C. nat\text{-of}\text{-fatom}\ (get\text{-atom}\ l) \leq n$
shows $eval_c\ HFun\ (extend\ f)\ C$

proof –

let $?F = HFun$

let $?G = extend\ f$

{

fix E

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

then have $\neg falsifies_g\ (f\ (Suc\ n))\ C$ **using** $instance\text{-of}_{I_s}\text{-self}$ **by** *auto*

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

blast

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

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

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

have $eval_l\ E\ HFun\ (extend\ f)\ l$

proof (*cases* l)

case ($Pos\ P\ ts$)

from $Pos\ l\text{-p}\ C\text{-ground}$ **have** $ts\text{-ground}: ground_{I_s}\ ts$ **by** *auto*

have $\neg falsifies_l\ (f\ (Suc\ n))\ l$ **using** $l\text{-p}$ **by** *auto*

then have $f\ (Suc\ n)\ !\ ?i = True$

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

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

then show $?thesis$ **using** $eval_1\text{-ground}_{ts}[of ts - ?G P] \text{ } ts\text{-ground } Pos$ **by**
 $auto$
next
case $(Neg P \text{ } ts)$
from $Neg \text{ } l\text{-}p \text{ } C\text{-ground}$ **have** $ts\text{-ground: } ground_{ts} \text{ } ts$ **by** $auto$

have $\neg falsifies_1 (f (Suc n)) \text{ } l$ **using** $l\text{-}p$ **by** $auto$
then have $f (Suc n) ! ?i = False$
using $j\text{-}n \text{ } Neg \text{ } ts\text{-ground} \text{ } empty\text{-subts}[of ts]$ **unfolding** $falsifies_1\text{-def}$ **by** $auto$
moreover have $f (Suc ?i) ! ?i = f (Suc n) ! ?i$
using $f\text{-infpath } i\text{-}n \text{ } j\text{-}n \text{ } \text{infpath-length}[of f] \text{ } ith\text{-in-extension}[of f]$ **by** $simp$
ultimately
have $f (Suc ?i) ! ?i = False$ **using** Neg **by** $auto$
then have $\neg ?G P (hterms\text{-of-}fterms \text{ } ts)$ **using** Neg **by** $(simp \text{ add: } nat\text{-of-fatom-def})$

then show $?thesis$ **using** $Neg \text{ } eval_1\text{-ground}_{ts}[of ts - ?G P] \text{ } ts\text{-ground}$ **by**
 $auto$
qed
then have $\exists l \in C. eval_1 \text{ } E \text{ } HFun (extend f) \text{ } l$ **using** $l\text{-}p$ **by** $auto$
}
then have $eval_c \text{ } HFun (extend f) \text{ } C$ **unfolding** $eval_c\text{-def}$ **by** $auto$
then show $?thesis$ **using** $instance\text{-of-}l_s\text{-self}$ **by** $auto$
qed

lemma $extend\text{-preserves-model2}$:

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

lemma $extend\text{-infpath}$:

assumes $f\text{-infpath: } wf\text{-infpath} (f :: nat \Rightarrow partial\text{-pred-denot})$
assumes $model\text{-}c: \forall n. \neg falsifies_c (f n) \text{ } C$

```

assumes fin-c: finite C
shows evalc HFun (extend f) C
unfolding evalc-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: ground1s (C ·1s sub-of-denot E) using sub-of-denot-equiv-ground
  by auto

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

```

assumes f-infnth: wf-infnth (f :: nat ⇒ partial-pred-denot)
assumes model-cs: ∀ n. ¬falsifiescs (f n) Cs
assumes fin-cs: finite Cs
assumes fin-c: ∀ C ∈ Cs. finite C
shows evalcs HFun (extend f) Cs
proof —
  let ?F = HFun

  have  $\forall C \in Cs. \text{eval}_c ?F (extend f) C$ 
  proof (rule ballI)
    fix C
    assume asm: C ∈ Cs
    then have  $\forall n. \neg \text{falsifies}_c (f n) C$  using model-cs by auto
    then show evalc ?F (extend f) C using fin-c asm f-infnth extend-infnth[of
    f C] by auto
  qed
  then show evalcs ?F (extend f) Cs unfolding evalcs-def by auto
qed

```

```

fun deeptree :: nat ⇒ tree where
  deeptree 0 = Leaf

```

| $deeptree (Suc\ n) = Branching\ (deeptree\ n)\ (deeptree\ n)$

lemma *branch-length*:

assumes *branch b (deeptree n)*

shows $length\ b = n$

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

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

next

case (*Suc n*)

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

then obtain *a b'* **where** $p: b = a\#\ b' \wedge$ *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-of-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** $\text{assms longer-falsifies}_l$ **by auto**
} then show $?thesis$ **using** assms **by auto**
qed

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

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

lemma longer-falsifies :
assumes $\text{falsifies}_{cs} ds Cs$
shows $\text{falsifies}_{cs} (ds @ d) Cs$
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** $\text{longer-falsifies}_c[\text{of } C ds d]$ **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 $\text{herbrand}'$:
assumes $\text{openb}: \forall T. \exists G. \text{open-branch } G T Cs$
assumes $\text{finite-cs}: \text{finite } Cs \forall C \in Cs. \text{finite } C$
shows $\exists G. \text{eval}_{cs} \text{HFun } G Cs$
proof –
– Show T infinite:
let $?tree = \{G. \neg \text{falsifies}_{cs} G Cs\}$
let $?undiaq = \text{length}$
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** $\text{diag-open}: \forall l. \text{open-branch } (?diag l) (\text{deeptree } l) Cs$
using $\text{someI-ex}[\text{of } \lambda b. \text{open-branch } b (\text{deeptree } -) Cs]$ **by auto**
then have $\forall n. ?undiaq (?diag n) = n$ **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** $\text{infinity}[\text{of } - \lambda n. \text{SOME } b. \text{open-branch } b (- n) Cs]$ **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 $\exists c. wf\text{-infpth } c \wedge (\forall n. c n \in ?tree)$ **using** *konig*[of *?tree*] **by** *blast*
then have $\exists G. wf\text{-infpth } G \wedge (\forall n. \neg \text{falsifies}_{cs} (G n) Cs)$ **by** *auto*
— Apply above infpath lemma:
then show $\exists G. eval_{cs} \text{ HFun } G Cs$ **using** *infpth-model finite-cs* **by** *auto*
qed

lemma *shorter-falsifies₁*:
assumes *falsifies₁* (*ds@d*) *l*
assumes *nat-of-fatom* (*get-atom l*) < *length ds*
shows *falsifies₁* *ds l*
proof —
let *?i* = *nat-of-fatom* (*get-atom l*)
from *assms* **have** *i-p*: *ground₁ l* \wedge *?i* < *length (ds@d)* \wedge (*ds@d*) ! *?i* = (\neg *sign*
l) **unfolding** *falsifies₁-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** *falsifies₁-def* **by** *simp*
qed

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

theorem *herbrand*:
assumes *unsat*: $\forall G. \neg eval_{cs} \text{ HFun } G Cs$
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 in *Unification_Theorem.thy* and used in *Completeness_Instance.thy*.

lemma *lifting*:

assumes *fin*: $\text{finite } C_1 \wedge \text{finite } C_2$

assumes *apart*: $\text{vars}_{l_s} C_1 \cap \text{vars}_{l_s} C_2 = \{\}$

assumes *inst*: $\text{instance-of}_{l_s} C_1' C_1 \wedge \text{instance-of}_{l_s} C_2' C_2$

assumes *appl*: $\text{applicable } C_1' C_2' L_1' L_2' \sigma$

shows $\exists L_1 L_2 \tau. \text{applicable } C_1 C_2 L_1 L_2 \tau \wedge$

$\text{instance-of}_{l_s} (\text{resolution } C_1' C_2' L_1' L_2' \sigma) (\text{resolution } C_1 C_2 L_1 L_2$

$\tau)$

proof –

— Obtaining the subsets we resolve upon:

let $?R_1' = C_1' - L_1'$ **and** $?R_2' = C_2' - L_2'$

from *inst* **obtain** $\gamma \mu$ **where** $C_1 \cdot l_s \gamma = C_1' \wedge C_2 \cdot l_s \mu = C_2'$

unfolding *instance-of_{l_s}*-def **by** *auto*

then obtain η **where** η -p: $C_1 \cdot l_s \eta = C_1' \wedge C_2 \cdot l_s \eta = C_2'$

using *apart merge-sub* **by** *force*

from η -p **obtain** L_1 **where** L_1 -p: $L_1 \subseteq C_1 \wedge L_1 \cdot l_s \eta = L_1' \wedge (C_1 - L_1) \cdot l_s \eta = ?R_1'$

using *appl project-sub* **using** *applicable-def* **by** *metis*

let $?R_1 = C_1 - L_1$

from η -p **obtain** L_2 **where** L_2 -p: $L_2 \subseteq C_2 \wedge L_2 \cdot l_s \eta = L_2' \wedge (C_2 - L_2) \cdot l_s \eta = ?R_2'$

using *appl project-sub* **using** *applicable-def* **by** *metis*

let $?R_2 = C_2 - L_2$

— Obtaining substitutions:

from *appl* **have** $\text{mgu}_{l_s} \sigma (L_1' \cup L_2'^C)$ **using** *applicable-def* **by** *auto*

then have $\text{mgu}_{l_s} \sigma ((L_1 \cdot l_s \eta) \cup (L_2 \cdot l_s \eta)^C)$ **using** L_1 -p L_2 -p **by** *auto*

then have $\text{mgu}_{l_s} \sigma ((L_1 \cup L_2^C) \cdot l_s \eta)$ **using** *compls-subls subls-union* **by** *auto*

then have $\text{unifier}_{l_s} \sigma ((L_1 \cup L_2^C) \cdot l_s \eta)$ **using** *mgu_{l_s}*-def **by** *auto*

then have $\eta \sigma \text{uni}: \text{unifier}_{l_s} (\eta \cdot \sigma) (L_1 \cup L_2^C)$

using *unifier_{l_s}*-def composition-conseq2l **by** *auto*

then obtain τ **where** τ -p: $\text{mgu}_{l_s} \tau (L_1 \cup L_2^C)$

using *unification fin L₁-p L₂-p* **by** (*meson finite-UnI finite-imageI rev-finite-subset*)

then obtain φ **where** φ -p: $\tau \cdot \varphi = \eta \cdot \sigma$ **using** $\eta \sigma \text{uni}$ *mgu_{l_s}*-def **by** *auto*

— Showing that we have the desired resolvent:

let $?C = ((C_1 - L_1) \cup (C_2 - L_2)) \cdot l_s \tau$

have $?C \cdot_{1s} \varphi = (?R_1 \cup ?R_2) \cdot_{1s} (\tau \cdot \varphi)$
using *subls-union composition-conseq2ls* **by** *auto*
also have $\dots = (?R_1 \cup ?R_2) \cdot_{1s} (\eta \cdot \sigma)$ **using** φ -*p* **by** *auto*
also have $\dots = ((?R_1 \cdot_{1s} \eta) \cup (?R_2 \cdot_{1s} \eta)) \cdot_{1s} \sigma$
using *subls-union composition-conseq2ls* **by** *auto*
also have $\dots = (?R_1' \cup ?R_2') \cdot_{1s} \sigma$ **using** η -*p* L_1 -*p* L_2 -*p* **by** *auto*
finally have $?C \cdot_{1s} \varphi = ((C_1' - L_1') \cup (C_2' - L_2')) \cdot_{1s} \sigma$ **by** *auto*
then have *ins: instance-of_{1s} (resolution C_{1'} C_{2'} L_{1'} L_{2'} σ) (resolution C₁ C₂ L₁ L₂ τ)*
using *resolution-def instance-of_{1s}-def* **by** *metis*

— Showing that the resolution rule is applicable:

have $C_1' \neq \{\} \wedge C_2' \neq \{\} \wedge L_1' \neq \{\} \wedge L_2' \neq \{\}$
using *appl applicable-def* **by** *auto*
then have $C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$ **using** η -*p* L_1 -*p* L_2 -*p* **by** *auto*
then have *appli: applicable C₁ C₂ L₁ L₂ τ*
using *apart L₁-p L₂-p τ-p applicable-def* **by** *auto*

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

17 Completeness

lemma *falsifies_g-empty*:
assumes *falsifies_g [] C*
shows $C = \{\}$

proof —

have $\forall l \in C. \text{False}$

proof

fix l

assume $l \in C$

then have *falsifies₁ [] l* **using** *assms* **by** *auto*

then show *False* **unfolding** *falsifies₁-def* **by** (*cases l*) *auto*

qed

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

qed

lemma *falsifies_{cs}-empty*:
assumes *falsifies_c [] C*
shows $C = \{\}$

proof —

from *assms* **obtain** C' **where** C' -*p*: *instance-of_{1s} C' C ∧ falsifies_g [] C'* **by** *auto*

then have $C' = \{\}$ **using** *falsifies_g-empty* **by** *auto*

then show $C = \{\}$ **using** C' -*p* **unfolding** *instance-of_{1s}-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  $falsif$ :  $falsifies_g G C_1'$ 
shows  $False$ 
proof ( $cases\ l_1$ )
  case ( $Pos\ p\ ts$ )
    let  $?i1 = nat-of-fatomb (p, ts)$ 

    from  $assms$  have  $gr$ :  $ground_l\ l_1$  unfolding  $falsifies_1-def$  by  $auto$ 
    then have  $Neg$ :  $l_2 = Neg\ p\ ts$  using  $comp\ Pos$  by ( $cases\ l_2$ )  $auto$ 

    from  $falsif$  have  $falsifies_l\ G\ l_1$  using  $l1C1'$  by  $auto$ 
    then have  $G\ !\ ?i1 = False$  using  $l1C1'\ Pos$  unfolding  $falsifies_1-def$  by ( $induction\ Pos\ p\ ts$ )  $auto$ 
    moreover
      let  $?i2 = nat-of-fatomb (get-atom\ l_2)$ 
      from  $falsif$  have  $falsifies_l\ G\ l_2$  using  $l2C1'$  by  $auto$ 
      then have  $G\ !\ ?i2 = (\neg sign\ l_2)$  unfolding  $falsifies_1-def$  by  $meson$ 
      then have  $G\ !\ ?i1 = (\neg sign\ l_2)$  using  $Pos\ Neg\ comp$  by  $simp$ 
      then have  $G\ !\ ?i1 = True$  using  $Neg$  by  $auto$ 
      ultimately show  $?thesis$  by  $auto$ 
    next
      case ( $Neg\ p\ ts$ )
        let  $?i1 = nat-of-fatomb (p,ts)$ 

        from  $assms$  have  $gr$ :  $ground_l\ l_1$  unfolding  $falsifies_1-def$  by  $auto$ 
        then have  $Pos$ :  $l_2 = Pos\ p\ ts$  using  $comp\ Neg$  by ( $cases\ l_2$ )  $auto$ 

        from  $falsif$  have  $falsifies_l\ G\ l_1$  using  $l1C1'$  by  $auto$ 
        then have  $G\ !\ ?i1 = True$  using  $l1C1'\ Neg$  unfolding  $falsifies_1-def$  by ( $metis\ get-atom.simps(2)\ literal.disc(2)$ )
        moreover
          let  $?i2 = nat-of-fatomb (get-atom\ l_2)$ 
          from  $falsif$  have  $falsifies_l\ G\ l_2$  using  $l2C1'$  by  $auto$ 
          then have  $G\ !\ ?i2 = (\neg sign\ l_2)$  unfolding  $falsifies_1-def$  by  $meson$ 
          then have  $G\ !\ ?i1 = (\neg sign\ l_2)$  using  $Pos\ Neg\ comp$  by  $simp$ 
          then have  $G\ !\ ?i1 = False$  using  $Pos$  using  $literal.disc(1)$  by  $blast$ 
          ultimately show  $?thesis$  by  $auto$ 
        qed

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

lemma  $other-falsified$ :

```

assumes $C_1'-p$: $ground_{l_s} C_1' \wedge falsifies_g (B@[d]) C_1'$
assumes $l-p$: $l \in C_1'$ *nat-of-fatom* ($get_atom\ l$) = $length\ B$
assumes *other*: $lo \in C_1'$ $lo \neq l$
shows $falsifies_l B\ lo$
proof –
let $?i = nat-of-fatom (get_atom\ lo)$
have $ground-l_2$: $ground_l\ l$ **using** $l-p\ C_1'-p$ **by** *auto*
– They are, of course, also ground:
have $ground-lo$: $ground_l\ lo$ **using** $C_1'-p$ *other* **by** *auto*
from $C_1'-p$ **have** $falsifies_g (B@[d]) (C_1' - \{l\})$ **by** *auto*
– And indeed, falsified by $B @ [d]$:
then have loB_2 : $falsifies_l (B@[d])\ lo$ **using** *other* **by** *auto*
then have $?i < length (B @ [d])$ **unfolding** $falsifies_l-def$ **by** *meson*
– And they have numbers in the range of $B @ [d]$, i.e. less than $length\ B + 1$:
then have $nat-of-fatom (get_atom\ lo) < length\ B + 1$ **using** *undia-diag-fatom*
by (*cases lo*) *auto*
moreover
have $l-lo$: $l \neq lo$ **using** *other* **by** *auto*
– They 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 $lo\ C_1'\ l$
($B@[d]$)] **by** *auto*
from $l-lo\ lc-lo$ **have** $get_atom\ l \neq get_atom\ lo$ **using** *sign-comp-atom* **by** *metis*
then have $nat-of-fatom (get_atom\ lo) \neq nat-of-fatom (get_atom\ l)$
using *nat-of-fatom-bij ground-lo ground-l_2 ground_l-ground-fatom*
unfolding *bij-betw-def inj-on-def* **by** *metis*
– Therefore they have different numbers:
then have $nat-of-fatom (get_atom\ lo) \neq length\ B$ **using** $l-p$ **by** *auto*
ultimately
– So their numbers are in the range of B :
have $nat-of-fatom (get_atom\ lo) < length\ B$ **by** *auto*
– So we did not need the last index of $B @ [d]$ to falsify them, i.e. B suffices:
then show $falsifies_l B\ lo$ **using** loB_2 *shorter-falsifies_l* **by** *blast*
qed

theorem completeness':
assumes *closed-tree T Cs*
assumes $\forall C \in Cs. finite\ C$
shows $\exists Cs'. resolution-deriv\ Cs\ Cs' \wedge \{\} \in Cs'$
using *assms proof* (*induction T arbitrary: Cs rule: measure-induct-rule*[of *tree-size*])
fix $T :: tree$
fix $Cs :: fterm\ clause\ set$
assume *ih*: $\bigwedge T' Cs. treesize\ T' < treesize\ T \implies closed-tree\ T'\ Cs \implies$
 $\forall C \in Cs. finite\ C \implies \exists Cs'. resolution-deriv\ Cs\ Cs' \wedge \{\} \in$
 Cs'
assume *clo*: *closed-tree T Cs*
assume *finite-Cs*: $\forall C \in Cs. finite\ C$
{ – Base case:
assume $treesize\ T = 0$

then have $T=Leaf$ **using** *treesize-Leaf* **by** *auto*
then have *closed-branch* \square *Leaf Cs* **using** *branch-inv-Leaf clo unfolding*
closed-tree-def **by** *auto*
then have *falsifies_{Cs}* \square *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 *treesize* $T > 0$
then have $\exists l r. T=Branching \ l \ r$ **by** (*cases T*) *auto*

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

obtain C_{1o} **where** *C_{1o-p}: C_{1o} \in Cs \wedge falsifies_c ?B₁ C_{1o}* **using** *b-p clo*
unfolding *closed-tree-def* **by** *metis*
obtain C_{2o} **where** *C_{2o-p}: C_{2o} \in Cs \wedge falsifies_c ?B₂ C_{2o}* **using** *b-p clo*
unfolding *closed-tree-def* **by** *metis*

— Standardizing the clauses apart:
let $?C_1 = \text{std}_1 \ C_{1o}$
let $?C_2 = \text{std}_2 \ C_{2o}$
have $C_{1-p}: \text{falsifies}_c \ ?B_1 \ ?C_1$ **using** *std₁-falsifies C_{1o-p}* **by** *auto*
have $C_{2-p}: \text{falsifies}_c \ ?B_2 \ ?C_2$ **using** *std₂-falsifies C_{2o-p}* **by** *auto*

have *fin: finite ?C₁ \wedge finite ?C₂* **using** *C_{1o-p} C_{2o-p} finite-Cs* **by** *auto*

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

— C_1' is falsified by $B @ [True]$:
from C_{1-p} **obtain** C_1' **where** *C_{1'-p}: ground_{l_s} C_{1'} \wedge instance-of_{l_s} C_{1'} ?C₁*
 \wedge *falsifies_g ?B₁ C_{1'}* **by** *metis*

have $\neg \text{falsifies}_c \ B \ C_{1o}$ **using** *C_{1o-p} b-p clo* **unfolding** *closed-tree-def* **by** *metis*
then have $\neg \text{falsifies}_c \ B \ ?C_1$ **using** *std₁-falsifies* **using** *prod.exhaust-sel* **by**
blast
— C_1' is not falsified by B :
then have $l-B: \neg \text{falsifies}_g \ B \ C_1'$ **using** $C_{1'-p}$ **by** *auto*

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

$\neg(\text{falsifies}_l B l_1)$ **by auto**
let $?i = \text{nat-of-fatom}(\text{get-atom } l_1)$

— l_1 is of course ground:
have $\text{ground-}l_1: \text{ground}_l l_1$ **using** $C_1'-p l_1-p$ **by auto**

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

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

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

— We do the same exercise for $C_2o \cdot_{l_s} (\lambda x. \varepsilon ("?" @ x))$, $C_2', B @ [False]$, l_2 :
from C_2-p **obtain** C_2' **where** $C_2'-p: \text{ground}_{l_s} C_2' \wedge \text{instance-of}_{l_s} C_2' ?C_2$
 $\wedge \text{falsifies}_g ?B_2 C_2'$ **by metis**

have $\neg \text{falsifies}_c B C_2o$ **using** $C_2o-p b-p \text{ clo}$ **unfolding** closed-tree-def **by metis**
then have $\neg \text{falsifies}_c B ?C_2$ **using** $\text{std}_2\text{-falsifies}$ **using** prod.exhaust-sel **by**
 blast
then have $l\text{-}B: \neg \text{falsifies}_g B C_2'$ **using** $C_2'-p$ **by auto**

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

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

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

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

— l_2 is negative:

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

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

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

from C_2' -p l_2 -no l_2 -p **have** B - 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 cisl $_1$: $l_2^c = l_1$

proof —

from l_1 -no l_2 -no *ground*- l_1 *ground*- l_2 **have** *get-atom* $l_1 = \text{get-atom } l_2$

using *nat-of-fatom-bij* *ground* $_1$ -*ground-fatom*

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

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

qed

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

using l_1 -p l_2 -p C_1' -p *ground* $_{l_1}$ -*vars* $_{l_1}$ l_2 cisl $_1$ *empty-comp2* **unfolding** *mgu* $_{l_1}$ -*def* *unifier* $_{l_1}$ -*def* **by** *auto*

— Lifting to get a resolvent of $C_1 @ \cdot_{l_1} (\lambda x. \varepsilon ("1" @ x))$ and $C_2 @ \cdot_{l_2} (\lambda x. \varepsilon ("2" @ x))$:

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

using *std-apart-apart* C_1' -p C_2' -p *lifting*[*of* $?C_1 ?C_2 C_1' C_2' \{l_1\} \{l_2\}$ *Resolution.ε*] *fin* **by** *auto*

— Defining the clause to be derived, the new clausal form and the new tree:

— We name the resolvent C .

obtain C **where** C -p: $C = \text{resolution } ?C_1 ?C_2 L_1 L_2 \tau$ **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 $B @ [\text{True}]$ and $B @ [\text{False}]$ 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* $_g B (\text{resolution } C_1' C_2' \{l_1\} \{l_2\}$ *Resolution.ε*) **unfolding** *resolution-def* *empty-subls* **by** *auto*

then have *falsifies*- C : *falsifies* $_c B C$ **using** C -p $L_1 L_2 \tau$ -p **by** *auto*

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

have $T''\text{-bran}$: *anybranch* T'' (λb . *closed-branch* b T'' $CsNext$)
proof (*rule allI*; *rule impI*)
 fix b
 assume br : *branch* b T''
 from br **have** $b = B \vee$ *branch* b T **using** *branch-delete* $T''\text{-p}$ **by** *auto*
 then show *closed-branch* b T'' $CsNext$
 proof
 assume $b=B$
 then show *closed-branch* b T'' $CsNext$ **using** *falsifies-C* br $CsNext\text{-p}$ **by**
auto
 next
 assume *branch* b T
 then show *closed-branch* b T'' $CsNext$ **using** *clo* br $T''\text{-p}$ $CsNext\text{-p}$
unfolding *closed-tree-def* **by** *auto*
 qed
 qed
then have $T''\text{-bran2}$: *anybranch* T'' (λb . *falsifies_{Cs}* b $CsNext$) **by** *auto*

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

obtain T' **where** $T'\text{-p}$: $T' =$ *cutoff* (λG . *falsifies_{Cs}* G $CsNext$) \square T'' **by** *auto*
have $T'\text{-smaller}$: *treesize* $T' <$ *treesize* T **using** *treesize-cutoff*[*of* λG . *falsifies_{Cs}* G $CsNext$ \square T''] $T''\text{-smaller}$ **unfolding** $T'\text{-p}$ **by** *auto*

from $T''\text{-bran2}$ **have** *anybranch* T' (λb . *falsifies_{Cs}* b $CsNext$) **using** *cutoff-branch*[*of* T'' λb . *falsifies_{Cs}* b $CsNext$] $T'\text{-p}$ **by** *auto*
then have $T'\text{-bran}$: *anybranch* T' (λb . *closed-branch* b T' $CsNext$) **by** *auto*
have $T'\text{-intr}$: *anyinternal* T' (λp . \neg *falsifies_{Cs}* p $CsNext$) **using** $T'\text{-p}$ *cutoff-internal*[*of* T'' λb . *falsifies_{Cs}* b $CsNext$] $T''\text{-bran2}$ **by** *blast*
have $T'\text{-closed}$: *closed-tree* T' $CsNext$ **using** $T'\text{-bran}$ $T'\text{-intr}$ **unfolding** *closed-tree-def* **by** *auto*
have *finite-CsNext*: $\forall C \in CsNext$. *finite* C **unfolding** $CsNext\text{-p}$ $C\text{-p}$ *resolution-def* **using** *finite-Cs* *fin* **by** *auto*

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

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

moreover

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

have *resolution-step* Cs ($Cs \cup \{?C_1\}$) **using** *std₁-renames* *standardize-apart* $C_{10\text{-p}}$ **by** (*metis* *Un-insert-right*)

moreover

have *resolution-step* ($Cs \cup \{?C_1\}$) ($Cs \cup \{?C_1\} \cup \{?C_2\}$) **using** *std₂-renames*[*of* C_{20}] *standardize-apart*[*of* C_{20} - $?C_2$] $C_{20\text{-p}}$ **by** *auto*

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

moreover
then have *resolution-step* $(Cs \cup \{?C_1, ?C_2\}) (Cs \cup \{?C_1, ?C_2\} \cup \{C\})$
using $L_1 L_2 \tau$ -*p resolution-rule*[*of* $?C_1$ $Cs \cup \{?C_1, ?C_2\}$ $?C_2$ $L_1 L_2 \tau$] **using**
C-p by auto
then have *resolution-step* $(Cs \cup \{?C_1, ?C_2\}) CsNext$ **using** *CsNext-p by*
(simp add: Un-commute)
ultimately
have *resolution-deriv* $Cs CsNext$ **unfolding** *resolution-deriv-def* **by auto**
}
— Combining the two derivations, we get the desired derivation from Cs of $\{\}$:
ultimately have *resolution-deriv* $Cs Cs''$ **unfolding** *resolution-deriv-def* **by**
auto
then have $\exists Cs'. \text{resolution-deriv } Cs Cs' \wedge \{\} \in Cs'$ **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

definition *E-conv* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ var-denot} \Rightarrow 'b \text{ var-denot}$ **where**
E-conv *b-of-a* $E \equiv \lambda x. (b\text{-of-a } (E x))$

definition *F-conv* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ fun-denot} \Rightarrow 'b \text{ fun-denot}$ **where**
F-conv *b-of-a* $F \equiv \lambda f bs. b\text{-of-a } (F f (\text{map } (\text{inv } b\text{-of-a}) bs))$

definition *G-conv* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ pred-denot} \Rightarrow 'b \text{ pred-denot}$ **where**
G-conv *b-of-a* $G \equiv \lambda p bs. (G p (\text{map } (\text{inv } b\text{-of-a}) bs))$

lemma *eval_t-bij*:

assumes *bij* $(b\text{-of-a}::'a \Rightarrow 'b)$
show $\text{eval}_t (E\text{-conv } b\text{-of-a } E) (F\text{-conv } b\text{-of-a } F) t = b\text{-of-a } (\text{eval}_t E F t)$
proof (*induction t*)
case $(Fun f ts)$
then have $\text{map } (\text{inv } b\text{-of-a} \circ \text{eval}_t (E\text{-conv } b\text{-of-a } E) (F\text{-conv } b\text{-of-a } F)) ts =$
 $\text{eval}_{ts} E F ts$
unfolding *E-conv-def* *F-conv-def*
using *assms* *bij-is-inj* **by fastforce**
then have $b\text{-of-a } (F f (\text{map } (\text{inv } b\text{-of-a} \circ \text{eval}_t (E\text{-conv } b\text{-of-a } E) ((F\text{-conv } b\text{-of-a } F))) ts)) =$
 $b\text{-of-a } (F f (\text{eval}_{ts} E F ts))$ **by metis**
then show *?case* **using** *assms* **unfolding** *E-conv-def* *F-conv-def* **by auto**


```

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

lemma evalts-bij:
  assumes bij (b-of-a::'a ⇒ 'b)
  shows G-conv b-of-a G p (evalts (E-conv b-of-a E) (F-conv b-of-a F) ts) = G p
(evalts E F ts)
  using assms using evalt-bij
proof -
  have map (inv b-of-a ∘ evalt (E-conv b-of-a E) (F-conv b-of-a F)) ts = evalts
E F ts
  using evalt-bij assms bij-is-inj by fastforce
  then show ?thesis
  by (metis (no-types) G-conv-def map-map)
qed

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

lemma evalc-bij:
  assumes bij (b-of-a::'a ⇒ 'b)
  shows evalc (F-conv b-of-a F) (G-conv b-of-a G) C = evalc F G C
proof -
  {
    fix E :: char list ⇒ 'b
    assume bij-b-of-a: bij b-of-a
    assume C-sat: ∀ E :: char list ⇒ 'a. ∃ l ∈ C. evall E F G l
    have E-p: E = E-conv b-of-a (E-conv (inv b-of-a) E)
    unfolding E-conv-def using bij-b-of-a
    using bij-betw-inv-into-right by fastforce
    have ∃ l ∈ C. evall (E-conv b-of-a (E-conv (inv b-of-a) E)) (F-conv b-of-a F)
(G-conv b-of-a G) l
    using evall-bij bij-b-of-a C-sat by blast
    then have ∃ l ∈ C. evall E (F-conv b-of-a F) (G-conv b-of-a G) l using E-p

```

```

by auto
}
then show ?thesis
  by (meson evall-bij assms evalc-def)
qed

lemma evalcs-bij:
  assumes bij (b-of-a :: 'a ⇒ 'b)
  shows evalcs (F-conv b-of-a F) (G-conv b-of-a G) Cs ⟷ evalcs F G Cs
  by (meson evalc-bij assms evalcs-def)

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

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

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

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

  then show ?thesis by auto
qed

```

lemma *infinite-hterms*: *infinite* (*UNIV* :: *hterm set*)
proof –
 let *?diago* = $\lambda n. \text{HFun } (\text{string-of-nat } n) \square$
 let *?undiago* = $\lambda a. \text{nat-of-string } (\text{case } a \text{ of } \text{HFun } f \text{ ts} \Rightarrow f)$
 have $\forall n. ?undiago (?diago n) = n$ **using** *nat-of-string-string-of-nat* **by** *auto*
moreover
 have $\forall n. ?diago n \in \text{UNIV}$ **by** *auto*
 ultimately **show** *infinite* (*UNIV* :: *hterm set*) **using** *infinity*[*of* *?undiago* *?diago* *UNIV*] **by** *simp*
qed

theorem *completeness-countable*:
 assumes *inf-uni*: *infinite* (*UNIV* :: (*'u* :: *countable*) *set*)
 assumes *finite-cs*: *finite* *Cs* $\forall C \in Cs. \text{finite } C$
 assumes *unsat*: $\forall (F :: 'u \text{ fun-denot}) (G :: 'u \text{ pred-denot}). \neg \text{eval}_{cs} F G Cs$
 shows $\exists Cs'. \text{resolution-deriv } Cs Cs' \wedge \{\} \in Cs'$
proof –
 have $\forall (F :: \text{hterm fun-denot}) (G :: \text{hterm pred-denot}). \neg \text{eval}_{cs} F G Cs$
proof (*rule*; *rule*)
 fix *F* :: *hterm fun-denot*
 fix *G* :: *hterm pred-denot*

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

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

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

theorem *completeness-nat*:
 assumes *finite-cs*: *finite* *Cs* $\forall C \in Cs. \text{finite } C$
 assumes *unsat*: $\forall (F :: \text{nat fun-denot}) (G :: \text{nat pred-denot}). \neg \text{eval}_{cs} F G Cs$
 shows $\exists Cs'. \text{resolution-deriv } Cs Cs' \wedge \{\} \in Cs'$
using *assms completeness-countable* **by** *blast*

end — unification locale

end

18 Examples

theory *Examples* **imports** *Resolution* **begin**

```

value Var "x"
value Fun "one" []
value Fun "mul" [Var "y", Var "y"]
value Fun "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []]

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

```

fun $F_{nat} :: nat\ fun\ denot$ **where**

```

   $F_{nat} f [n, m] =$ 
    (if  $f = "add"$  then  $n + m$  else
     if  $f = "mul"$  then  $n * m$  else 0)
|  $F_{nat} f [] =$ 
  (if  $f = "one"$  then 1 else
   if  $f = "zero"$  then 0 else 0)
|  $F_{nat} f us = 0$ 

```

fun $G_{nat} :: nat\ pred\ denot$ **where**

```

   $G_{nat} p [x, y] =$ 
    (if  $p = "less" \wedge x < y$  then True else
     if  $p = "greater" \wedge x > y$  then True else
     if  $p = "equals" \wedge x = y$  then True else False)
|  $G_{nat} p us = False$ 

```

fun $E_{nat} :: nat\ var\ denot$ **where**

```

   $E_{nat} x =$ 
    (if  $x = "x"$  then 26 else
     if  $x = "y"$  then 5 else 0)

```

lemma $eval_t E_{nat} F_{nat} (Var "x") = 26$

by *auto*

lemma $eval_t E_{nat} F_{nat} (Fun "one" []) = 1$

by *auto*

lemma $eval_t E_{nat} F_{nat} (Fun "mul" [Var "y", Var "y"]) = 25$

by *auto*

lemma

$eval_t E_{nat} F_{nat} (Fun "add" [Fun "mul" [Var "y", Var "y"], Fun "one" []]) = 26$

by *auto*

lemma $eval_l E_{nat} F_{nat} G_{nat} (Pos "greater" [Var "x", Var "y"]) = True$

by *auto*

lemma $eval_l E_{nat} F_{nat} G_{nat} (Neg "less" [Var "x", Var "y"]) = True$

by *auto*

lemma $eval_l E_{nat} F_{nat} G_{nat} (Pos "less" [Var "x", Var "y"]) = False$

by auto

lemma *eval_l E_{nat} F_{nat} G_{nat}*
 (*Pos "equals"*
 [*Fun "add"* [*Fun "mul"* [*Var "y"*, *Var "y'*], *Fun "one"* []]
 , *Var "x'*]
] = *True*
)
 by auto

definition *PP* :: *fterm literal* **where**
 PP = *Pos "P"* [*Fun "c"* []]

definition *PQ* :: *fterm literal* **where**
 PQ = *Pos "Q"* [*Fun "d"* []]

definition *NP* :: *fterm literal* **where**
 NP = *Neg "P"* [*Fun "c"* []]

definition *NQ* :: *fterm literal* **where**
 NQ = *Neg "Q"* [*Fun "d"* []]

theorem *empty-mgu*:
 assumes *unifier_{l_s}* ∈ *L*
 shows *mgu_{l_s}* ∈ *L*
 using *assms unfolding unifier_{l_s}*-def *mgu_{l_s}*-def **apply** *auto*
 apply (*rule-tac x=u in exI*)
 using *empty-comp1 empty-comp2* **apply** *auto*
 done

theorem *unifier-single*: *unifier_{l_s}* σ {l}

unfolding *unifier_{l_s}*-def **by** *auto*

theorem *resolution-rule'*:
 assumes *C₁ ∈ Cs*
 assumes *C₂ ∈ Cs*
 assumes *applicable C₁ C₂ L₁ L₂ σ*
 assumes *C = {resolution C₁ C₂ L₁ L₂ σ}*
 shows *resolution-step Cs (Cs ∪ C)*
 using *assms resolution-rule* **by** *auto*

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

proof –

have *resolution-step*
 {{*NP, PQ*}, {*NQ*}, {*PP, PQ*}}
 ({{*NP, PQ*}, {*NQ*}, {*PP, PQ*}} ∪ {{*NP*}})
 apply (*rule resolution-rule'*[of {*NP, PQ*} - {*NQ*} {*PQ*} {*NQ*} ε])
 unfolding *applicable-def vars_{l_s}*-def vars_l-def

NQ-def NP-def PQ-def PP-def resolution-def
using *unifier-single empty-mgu using empty-subls*
apply *auto*
done
then have *resolution-step*
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}$
 $(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}\})$
by (*simp add: insert-commute*)
moreover
have *resolution-step*
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}\}$
 $(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}\} \cup \{\{PP\}\})$
apply (*rule resolution-rule* [of $\{NQ\}$ - $\{PP, PQ\}$ $\{NQ\}$ $\{PQ\}$ ε])
unfolding *applicable-def vars_{1s}-def vars₁-def*
NQ-def NP-def PQ-def PP-def resolution-def
using *unifier-single empty-mgu empty-subls* **apply** *auto*
done
then have *resolution-step*
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}\}$
 $(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}\})$
by (*simp add: insert-commute*)
moreover
have *resolution-step*
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}\}$
 $(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}\} \cup \{\{\}\})$
apply (*rule resolution-rule* [of $\{NP\}$ - $\{PP\}$ $\{NP\}$ $\{PP\}$ ε])
unfolding *applicable-def vars_{1s}-def vars₁-def*
NQ-def NP-def PQ-def PP-def resolution-def
using *unifier-single empty-mgu* **apply** *auto*
done
then have *resolution-step*
 $\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}\}$
 $(\{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}, \{NP\}, \{PP\}, \{\}\})$
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 resolution-example1-sem:  $\neg \text{eval}_{cs} F G \{\{NP, PQ\}, \{NQ\}, \{PP, PQ\}\}$ 
  using resolution-example1 derivation-sound-refute by auto

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

end

```

19 The Unification Theorem

theory *Unification-Theorem* **imports**

First-Order-Terms.Unification Resolution

begin

definition *set-to-list* :: 'a set \Rightarrow 'a list **where**

set-to-list \equiv *inv set*

lemma *set-set-to-list*: *finite xs* \implies *set (set-to-list xs) = xs*

proof (*induction rule: finite.induct*)

case (*emptyI*)

have *set [] = {}* **by** *auto*

then show *?case* **unfolding** *set-to-list-def inv-into-def* **by** *auto*

next

case (*insertI A a*)

then have *set (a#set-to-list A) = insert a A* **by** *auto*

then show *?case* **unfolding** *set-to-list-def inv-into-def* **by** (*metis (mono-tags, lifting) UNIV-I someI*)

qed

fun *iterm-to-fterm* :: (*fun-sym, var-sym*) *term* \Rightarrow *fterm* **where**

iterm-to-fterm (*Term.Var x*) = *Var x*

| *iterm-to-fterm* (*Term.Fun f ts*) = *Fun f (map iterm-to-fterm ts)*

fun *fterm-to-iterm* :: *fterm* \Rightarrow (*fun-sym, var-sym*) *term* **where**

fterm-to-iterm (*Var x*) = *Term.Var x*

| *fterm-to-iterm* (*Fun f ts*) = *Term.Fun f (map fterm-to-iterm ts)*

lemma *iterm-to-fterm-cancel[simp]*: *iterm-to-fterm (fterm-to-iterm t) = t*

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

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

abbreviation(*input*) *fsub-to-isub* :: *substitution* \Rightarrow (*fun-sym, var-sym*) *subst* **where**
fsub-to-isub $\sigma \equiv \lambda x. \text{fterm-to-iterm } (\sigma \ x)$

abbreviation(*input*) *isub-to-fsub* :: (*fun-sym, var-sym*) *subst* \Rightarrow *substitution* **where**
isub-to-fsub $\sigma \equiv \lambda x. \text{iterm-to-fterm } (\sigma \ x)$

lemma *iterm-to-fterm-subst*: (*iterm-to-fterm t1*) $\cdot_t \sigma = \text{iterm-to-fterm } (t1 \cdot (\lambda x. \text{fterm-to-iterm } (\sigma \ x)))$
by (*induction t1*) *auto*

lemma *unifiert-unifiers*:
assumes *unifier_{ts} σ ts*
shows *fsub-to-isub* $\sigma \in \text{unifiers } (\text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts)$
proof –
have $\forall t1 \in \text{fterm-to-iterm } 'ts. \forall t2 \in \text{fterm-to-iterm } 'ts. t1 \cdot (\text{fsub-to-isub } \sigma) = t2 \cdot (\text{fsub-to-isub } \sigma)$
proof (*rule ballI; rule ballI*)
fix *t1 t2*
assume *t1-p: t1* $\in \text{fterm-to-iterm } 'ts$ **assume** *t2-p: t2* $\in \text{fterm-to-iterm } 'ts$
from *t1-p t2-p* **have** *iterm-to-fterm t1* $\in ts \wedge \text{iterm-to-fterm } t2 \in ts$ **by** *auto*

then have (*iterm-to-fterm t1*) $\cdot_t \sigma = (\text{iterm-to-fterm } t2) \cdot_t \sigma$ **using** *assms*
unfolding *unifier_{ts}-def* **by** *auto*
then have *iterm-to-fterm (t1* $\cdot \text{fsub-to-isub } \sigma) = \text{iterm-to-fterm } (t2 \cdot \text{fsub-to-isub } \sigma)$ **using** *iterm-to-fterm-subst* **by** *auto*
then have *fterm-to-iterm (iterm-to-fterm (t1* $\cdot \text{fsub-to-isub } \sigma)) = \text{fterm-to-iterm } (iterm-to-fterm (t2 \cdot \text{fsub-to-isub } \sigma))$ **by** *auto*
then show *t1* $\cdot \text{fsub-to-isub } \sigma = t2 \cdot \text{fsub-to-isub } \sigma$ **using** *fterm-to-iterm-cancel*
by *auto*
qed
then have $\forall p \in \text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts. \text{fst } p \cdot \text{fsub-to-isub } \sigma = \text{snd } p \cdot \text{fsub-to-isub } \sigma$ **by** (*metis mem-Times-iff*)
then show *?thesis* **unfolding** *unifiers-def* **by** *blast*
qed

abbreviation(*input*) *get-mgut* :: *fterm list* \Rightarrow *substitution option* **where**
get-mgut *ts* $\equiv \text{map-option } (\text{isub-to-fsub } \circ \text{subst-of}) (\text{unify } (\text{List.product } (\text{map } \text{fterm-to-iterm } ts) (\text{map } \text{fterm-to-iterm } ts))) \ \square$

lemma *unify-unification*:
assumes $\sigma \in \text{unifiers } (\text{set } E)$
shows $\exists \vartheta. \text{is-ingu } \vartheta (\text{set } E)$
proof –
from *assms* **have** $\exists cs. \text{unify } E \ \square = \text{Some } cs$ **using** *unify-complete* **by** *auto*

then show *?thesis* **using** *unify-sound* **by** *auto*
qed

lemma *fterm-to-iterm-subst*: $(\text{fterm-to-iterm } t1) \cdot \sigma = \text{fterm-to-iterm } (t1 \cdot_t \text{isub-to-fsub } \sigma)$
by *(induction t1) auto*

lemma *unifiers-unifiert*:

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

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

proof *(cases ts={})*

assume $ts = \{\}$

then show $\text{unifier}_{ts} (\text{isub-to-fsub } \sigma) ts$ **unfolding** *unifier_{ts}-def* **by** *auto*

next

assume $ts \neq \{\}$

then obtain t' **where** $t'-p$: $t' \in ts$ **by** *auto*

have $\forall t_1 \in ts. \forall t_2 \in ts. t_1 \cdot_t \text{isub-to-fsub } \sigma = t_2 \cdot_t \text{isub-to-fsub } \sigma$

proof *(rule ballI ; rule ballI)*

fix $t_1 t_2$

assume $t_1 \in ts t_2 \in ts$

then have $\text{fterm-to-iterm } t_1 \in \text{fterm-to-iterm } 'ts \text{fterm-to-iterm } t_2 \in \text{fterm-to-iterm } 'ts$ **by** *auto*

then have $(\text{fterm-to-iterm } t_1, \text{fterm-to-iterm } t_2) \in (\text{fterm-to-iterm } 'ts \times \text{fterm-to-iterm } 'ts)$ **by** *auto*

then have $(\text{fterm-to-iterm } t_1) \cdot \sigma = (\text{fterm-to-iterm } t_2) \cdot \sigma$ **using** *assms* **unfolding** *unifiers-def*

by *(metis (no-types, lifting) assms fst-conv in-unifiersE snd-conv)*

then have $\text{fterm-to-iterm } (t_1 \cdot_t \text{isub-to-fsub } \sigma) = \text{fterm-to-iterm } (t_2 \cdot_t \text{isub-to-fsub } \sigma)$ **using** *fterm-to-iterm-subst* **by** *auto*

then have $\text{iterm-to-fterm } (\text{fterm-to-iterm } (t_1 \cdot_t (\text{isub-to-fsub } \sigma))) = \text{iterm-to-fterm } (\text{fterm-to-iterm } (t_2 \cdot_t \text{isub-to-fsub } \sigma))$ **by** *auto*

then show $t_1 \cdot_t \text{isub-to-fsub } \sigma = t_2 \cdot_t \text{isub-to-fsub } \sigma$ **by** *auto*

qed

then have $\forall t_2 \in ts. t' \cdot_t \text{isub-to-fsub } \sigma = t_2 \cdot_t \text{isub-to-fsub } \sigma$ **using** $t'-p$ **by** *blast*

then show $\text{unifier}_{ts} (\text{isub-to-fsub } \sigma) ts$ **unfolding** *unifier_{ts}-def* **by** *metis*

qed

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

unfolding *composition-def subst-compose-def*

proof

fix x

show $\vartheta x \cdot i = \text{fterm-to-iterm } (\text{iterm-to-fterm } (\vartheta x) \cdot_t (\lambda x. \text{iterm-to-fterm } (i x)))$ **using** *iterm-to-fterm-subt* **by** *auto*

qed

lemma *is-mgu-mgu_{ts}*:

assumes *finite ts*
assumes *is-imgu ϑ (fterm-to-iterm ' ts × fterm-to-iterm ' ts)*
shows *mgu_{ts} (isub-to-fsub ϑ) ts*
proof –
from *assms* **have** *unifier_{ts} (isub-to-fsub ϑ) ts* **unfolding** *is-imgu-def* **using**
unifiers-unifiert **by** *auto*
moreover **have** $\forall u. \text{unifier}_{ts} u ts \longrightarrow (\exists i. u = (\text{isub-to-fsub } \vartheta) \cdot i)$
proof (*rule allI; rule impI*)
fix *u*
assume *unifier_{ts} u ts*
then **have** *fsub-to-isub u ∈ unifiers (fterm-to-iterm ' ts × fterm-to-iterm ' ts)* **using**
unifiert-unifiers **by** *auto*
then **have** $\exists i. \text{fsub-to-isub } u = \vartheta \circ_s i$ **using** *assms* **unfolding** *is-imgu-def*
by *auto*
then **obtain** *i* **where** *fsub-to-isub u = $\vartheta \circ_s i$* **by** *auto*
then **have** *fsub-to-isub u = fsub-to-isub (isub-to-fsub $\vartheta \cdot \text{isub-to-fsub } i)$* **using**
icomf-fcomp **by** *auto*
then **have** *isub-to-fsub (fsub-to-isub u) = isub-to-fsub (fsub-to-isub (isub-to-fsub $\vartheta \cdot \text{isub-to-fsub } i)$)* **by** *metis*
then **have** *u = isub-to-fsub $\vartheta \cdot \text{isub-to-fsub } i$* **by** *auto*
then **show** $\exists i. u = \text{isub-to-fsub } \vartheta \cdot i$ **by** *metis*
qed
ultimately **show** *?thesis* **unfolding** *mgu_{ts}-def* **by** *auto*
qed

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

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

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

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

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

lemma *unifier_{l_s}-unifier_{t_s}*:

assumes *unifier_{l_s}* σ *L*

shows *unifier_{t_s}* σ (*literal-to-term* ‘ *L*)

proof –

from *assms* **obtain** *l'* **where** $\forall l \in L. l \cdot_l \sigma = l'$ **unfolding** *unifier_{l_s}-def* **by** *auto*

then have $\forall l \in L. \text{literal-to-term } (l \cdot_l \sigma) = \text{literal-to-term } l'$ **by** *auto*

then have $\forall l \in L. (\text{literal-to-term } l) \cdot_t \sigma = \text{literal-to-term } l'$ **using** *literal-to-term-sub*
by *auto*

then have $\forall t \in \text{literal-to-term ' } L. t \cdot_t \sigma = \text{literal-to-term } l'$ **by** *auto*

then show *?thesis* **unfolding** *unifier_{t_s}-def* **by** *auto*

qed

lemma *unifiert-unifier_{l_s}*:

assumes *unifier_{t_s}* σ (*literal-to-term* ‘ *L*)

shows *unifier_{l_s}* σ *L*

proof –

from *assms* **obtain** *t'* **where** $\forall t \in \text{literal-to-term ' } L. t \cdot_t \sigma = t'$ **unfolding**
unifier_{t_s}-def **by** *auto*

then have $\forall t \in \text{literal-to-term ' } L. \text{term-to-literal } (t \cdot_t \sigma) = \text{term-to-literal } t'$ **by**
auto

then have $\forall l \in L. \text{term-to-literal } ((\text{literal-to-term } l) \cdot_t \sigma) = \text{term-to-literal } t'$ **by**
auto

then have $\forall l \in L. \text{term-to-literal } ((\text{literal-to-term } (l \cdot_l \sigma))) = \text{term-to-literal } t'$
using *literal-to-term-sub* **by** *auto*

then have $\forall l \in L. l \cdot_l \sigma = \text{term-to-literal } t'$ **by** *auto*

then show *?thesis* **unfolding** *unifier_{l_s}-def* **by** *auto*

qed

lemma *mgu_{t_s}-mgu_{l_s}*:

assumes *mgu_{t_s}* ϑ (*literal-to-term* ‘ *L*)

shows *mgu_{l_s}* ϑ *L*

proof –

from *assms* **have** *unifier_{t_s}* ϑ (*literal-to-term* ‘ *L*) **unfolding** *mgu_{t_s}-def* **by** *auto*

then have *unifier_{l_s}* ϑ *L* **using** *unifiert-unifier_{l_s}* **by** *auto*

moreover

{

fix *u*

assume *unifier_{l_s}* *u* *L*

then have *unifier_{t_s}* *u* (*literal-to-term* ‘ *L*) **using** *unifier_{l_s}-unifier_{t_s}* **by** *auto*

then have $\exists i. u = \vartheta \cdot i$ **using** *assms* **unfolding** *mgu_{t_s}-def* **by** *auto*

}

ultimately show *?thesis* **unfolding** *mgu_{l_s}-def* **by** *auto*

qed

theorem *unification*:
assumes *fin*: *finite L*
assumes *uni*: *unifier_{l_s} σ L*
shows $\exists \vartheta. \text{mgu}_{l_s} \vartheta L$
proof –
from *uni* **have** *unifier_{t_s} σ* (*literal-to-term ‘ L*) **using** *unifier_{l_s}-unifier_{t_s}* **by** *auto*
then obtain ϑ **where** *mgu_{t_s} ϑ* (*literal-to-term ‘ L*) **using** *fin unification'* **by**
blast
then have *mgu_{l_s} ϑ L* **using** *mgu_{t_s}-mgu_{l_s}* **by** *auto*
then show *?thesis* **by** *auto*
qed
end

20 Instance of completeness theorem

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

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

thm *lifting*

lemma *lift*:

assumes *fin*: *finite C ∧ finite D*
assumes *apart*: *vars_{l_s} C ∩ vars_{l_s} D = {}*
assumes *inst₁*: *instance-of_{l_s} C' C*
assumes *inst₂*: *instance-of_{l_s} D' D*
assumes *appl*: *applicable C' D' L' M' σ*
shows $\exists L M \tau. \text{applicable } C D L M \tau \wedge$
 $\text{instance-of}_{l_s} (\text{resolution } C' D' L' M' \sigma) (\text{resolution } C D L M \tau)$
using *assms lifting* **by** *metis*

thm *completeness*

theorem *complete*:

assumes *finite-cs*: *finite Cs ∨ C ∈ Cs. finite C*
assumes *unsat*: $\forall (F::\text{hterm fun-denot}) (G::\text{hterm pred-denot}). \neg \text{eval}_{cs} F G Cs$
shows $\exists Cs'. \text{resolution-deriv } Cs Cs' \wedge \{\} \in Cs'$
using *assms completeness* **by** –

thm *completeness-countable*

theorem *complete-countable*:

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

thm *completeness-nat*

theorem *complete-nat*:

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

assumes *unsat*: $\forall (F::nat\ fun\ denot) (G::nat\ pred\ denot) . \neg eval_{Cs} F G Cs$

shows $\exists Cs'$. *resolution-deriv Cs Cs' $\wedge \{\} \in Cs'$*

using *assms completeness-nat* **by** –

end

References

- [1] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 3rd edition, 2012.
- [2] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1st edition, 1973.
- [3] IsaFoL authors. IsaFoL: Isabelle Formalization of Logic. <https://bitbucket.org/isafol/isafol>.
- [4] A. Leitsch. *The Resolution Calculus*. Texts in theoretical computer science. Springer, 1997.
- [5] A. Schlichtkrull. Formalization of resolution calculus in Isabelle. Msc thesis, Technical University of Denmark, 2015. <https://people.compute.dtu.dk/andschl/Thesis.pdf>.
- [6] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. In *ITP 2016*, volume 9807 of *LNCS*. Springer, 2016.
- [7] A. Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 2018.