

A Verified Decision Procedure for a Quantifier-Free
Fragment of Set Theory

Lukas Stevens

February 6, 2026

Abstract

This formalization verifies a decision procedure due to Cantone and Zarba for a quantifier-free fragment of set theory. The fragment is called multi-level syllogistic with singleton, or MLSS for short. Its syntax includes the usual set operations union, intersection, difference, membership, equality as well as the construction of a set containing a single element. We specify the semantics of MLSS in terms of hereditarily finite sets and provide a sound and complete tableau calculus for it. We also provide an executable specification of a decision procedure that applies the rules of the calculus exhaustively and prove its termination. Furthermore, we extend the calculus with a light-weight type system that paves the way for an integration of the procedure into Isabelle/HOL.

Contents

1	Syntax and Semantics of MLSS	2
1.1	Propositional formulae	2
1.2	Definition of MLSS	3
1.2.1	Syntax and Semantics	3
1.2.2	Variables	4
1.2.3	Subformulae and Subterms	4
1.2.4	Finiteness of Variables, Subterms, and Subformulae	9
1.2.5	Non-Emptiness of Subterms	10
2	A Tableau Calculus for MLSS	11
2.1	Typing Rules	11
2.2	The Calculus	12
2.2.1	Closedness	12
2.2.2	Linear Expansion Rules	12
2.2.3	Branching Expansion Rules	16
2.2.4	Expansion Closure	18
2.2.5	Well-Formed Branch	19
3	A Type Checker for MLSS	20
3.1	Solver for the Theory of the Successor Function	20
3.2	Typing and Branch Expansion	30
3.3	Typing the Urelements	40
4	Deciding MLSS	46
4.1	The Realisation Function	46
4.2	A Decision Procedure for MLSS	51
4.2.1	Basic Definitions	51
4.2.2	Completeness of the Calculus	57
4.2.3	Soundness of the Calculus	95
4.2.4	Upper Bounding the Cardinality of a Branch	100
4.2.5	The Decision Procedure	106
4.3	An Executable Specification of the Procedure	110

Chapter 1

Syntax and Semantics of MLSS

1.1 Propositional formulae

datatype (*atoms: 'a*) *fm* =
 is-Atom: Atom 'a | *And 'a fm 'a fm* | *Or 'a fm 'a fm* |
 Neg 'a fm

fun *interp* :: (*'model* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow *'model* \Rightarrow *'a fm* \Rightarrow *bool* **where**
 interp I_a M (Atom a) = I_a M a |
 interp I_a M (And φ_1 φ_2) = (interp I_a M φ_1 \wedge interp I_a M φ_2) |
 interp I_a M (Or φ_1 φ_2) = (interp I_a M φ_1 \vee interp I_a M φ_2) |
 interp I_a M (Neg φ) = (\neg interp I_a M φ)

locale *ATOM* =
 fixes *I_a* :: *'model* \Rightarrow *'a* \Rightarrow *bool*
begin

abbreviation *I* **where** *I* \equiv *interp I_a*

end

definition *Atoms A* \equiv {*a* | *a. Atom a* \in *A*}

lemma *Atoms-Un[simp]*: *Atoms (A* \cup *B) = Atoms A* \cup *Atoms B*
 unfolding *Atoms-def* **by** *auto*

lemma *Atoms-mono*: *A* \subseteq *B* \implies *Atoms A* \subseteq *Atoms B*
 unfolding *Atoms-def* **by** *auto*

1.2 Definition of MLSS

Here, we define the syntax and semantics of multi-level syllogistic with singleton (MLSS). Additionally, we define a number of functions working on the syntax such as a function that collects all the subterms of a term.

1.2.1 Syntax and Semantics

datatype (*vars-term*: 'a) *pset-term* =
Empty nat | *is-Var*: Var 'a |
Union 'a *pset-term* 'a *pset-term* |
Inter 'a *pset-term* 'a *pset-term* |
Diff 'a *pset-term* 'a *pset-term* |
Single 'a *pset-term*

datatype (*vars-atom*: 'a) *pset-atom* =
Elem 'a *pset-term* 'a *pset-term* |
Equal 'a *pset-term* 'a *pset-term*

open-bundle *mlss-syntax*

begin

notation *Empty* ($\langle \emptyset \ - \rangle$)

notation *Union* (**infixr** $\langle \sqcup_s \rangle$ 165)

notation *Inter* (**infixr** $\langle \sqcap_s \rangle$ 170)

notation *Diff* (**infixl** $\langle -_s \rangle$ 180)

notation *Elem* (**infix** $\langle \in_s \rangle$ 150)

notation *Equal* (**infix** $\langle =_s \rangle$ 150)

end

abbreviation *AT* $a \equiv \text{Atom } a$

abbreviation *AF* $a \equiv \text{Neg } (\text{Atom } a)$

type-synonym 'a *pset-fm* = 'a *pset-atom fm*

type-synonym 'a *branch* = 'a *pset-fm list*

fun *I_{st}* :: ('a \Rightarrow hf) \Rightarrow 'a *pset-term* \Rightarrow hf **where**

$I_{st} \ v \ (\emptyset \ n) = 0$
 $| \ I_{st} \ v \ (\text{Var } x) = v \ x$
 $| \ I_{st} \ v \ (s1 \ \sqcup_s \ s2) = I_{st} \ v \ s1 \ \sqcup \ I_{st} \ v \ s2$
 $| \ I_{st} \ v \ (s1 \ \sqcap_s \ s2) = I_{st} \ v \ s1 \ \sqcap \ I_{st} \ v \ s2$
 $| \ I_{st} \ v \ (s1 \ -_s \ s2) = I_{st} \ v \ s1 \ - \ I_{st} \ v \ s2$
 $| \ I_{st} \ v \ (\text{Single } s) = HF \ \{I_{st} \ v \ s\}$

fun *I_{sa}* :: ('a \Rightarrow hf) \Rightarrow 'a *pset-atom* \Rightarrow bool **where**

$I_{sa} \ v \ (t1 \ \in_s \ t2) \ \longleftrightarrow \ I_{st} \ v \ t1 \ \in \ I_{st} \ v \ t2$
 $| \ I_{sa} \ v \ (t1 \ =_s \ t2) \ \longleftrightarrow \ I_{st} \ v \ t1 = I_{st} \ v \ t2$

1.2.2 Variables

definition *vars-fm* :: 'a pset-fm \Rightarrow 'a set **where**
vars-fm $\varphi \equiv \bigcup(\text{vars-atom } ' \text{ atoms } \varphi)$

definition *vars-branch* :: 'a branch \Rightarrow 'a set **where**
vars-branch $b \equiv \bigcup(\text{vars-fm } ' \text{ set } b)$

consts *vars* :: 'b \Rightarrow 'a set

adhoc-overloading

vars \equiv *vars-term* **and**

vars \equiv *vars-atom* **and**

vars \equiv *vars-fm* **and**

vars \equiv *vars-branch*

lemma *vars-fm-simps[simp]*:

vars (Atom a) = *vars* a

vars (And p q) = *vars* $p \cup$ *vars* q

vars (Or p q) = *vars* $p \cup$ *vars* q

vars (Neg p) = *vars* p

unfolding *vars-fm-def*

apply(*auto*)

done

lemma *vars-fmI*:

$x \in \text{vars } p \implies x \in \text{vars } (\text{And } p \ q)$

$x \in \text{vars } q \implies x \in \text{vars } (\text{And } p \ q)$

$x \in \text{vars } p \implies x \in \text{vars } (\text{Or } p \ q)$

$x \in \text{vars } q \implies x \in \text{vars } (\text{Or } p \ q)$

$x \in \text{vars } p \implies x \in \text{vars } (\text{Neg } p)$

by *auto*

lemma *vars-branch-simps*:

vars [] = {}

vars ($x \#$ xs) = *vars* $x \cup$ *vars* xs

unfolding *vars-branch-def* **by** *auto*

lemma *vars-branch-append*:

vars ($b1$ @ $b2$) = *vars* $b1 \cup$ *vars* $b2$

unfolding *vars-branch-def* **by** *simp*

lemma *vars-fm-vars-branchI*:

$\varphi \in \text{set } b \implies x \in \text{vars-fm } \varphi \implies x \in \text{vars-branch } b$

unfolding *vars-branch-def* **by** *blast*

1.2.3 Subformulae and Subterms

Subformulae

fun *subfms* :: 'a fm \Rightarrow 'a fm set **where**

$subfms (Atom a) = \{Atom a\}$
 $| subfms (And p q) = \{And p q\} \cup subfms p \cup subfms q$
 $| subfms (Or p q) = \{Or p q\} \cup subfms p \cup subfms q$
 $| subfms (Neg q) = \{Neg q\} \cup subfms q$

definition *subfms-branch* :: 'a fm list \Rightarrow 'a fm set **where**
subfms-branch b $\equiv \bigcup (subfms \text{ ' set } b)$

lemma *subfms-branch-simps*:

$subfms\text{-branch } [] = \{\}$
 $subfms\text{-branch } (x \# xs) = subfms x \cup subfms\text{-branch } xs$
unfolding *subfms-branch-def* **by** *auto*

lemma *subfms-refl[simp]*: $p \in subfms p$
by (*cases p*) *auto*

lemma *subfmsI*:

$a \in subfms p \Longrightarrow a \in subfms (And p q)$
 $a \in subfms q \Longrightarrow a \in subfms (And p q)$
 $a \in subfms p \Longrightarrow a \in subfms (Or p q)$
 $a \in subfms q \Longrightarrow a \in subfms (Or p q)$
 $a \in subfms p \Longrightarrow a \in subfms (Neg p)$
by *auto*

lemma *subfms-trans*: $q \in subfms p \Longrightarrow p \in subfms r \Longrightarrow q \in subfms r$
by (*induction r*) *auto*

lemma *subfmsD*:

$And p q \in subfms \varphi \Longrightarrow p \in subfms \varphi$
 $And p q \in subfms \varphi \Longrightarrow q \in subfms \varphi$
 $Or p q \in subfms \varphi \Longrightarrow p \in subfms \varphi$
 $Or p q \in subfms \varphi \Longrightarrow q \in subfms \varphi$
 $Neg p \in subfms \varphi \Longrightarrow p \in subfms \varphi$
using *subfmsI subfms-refl subfms-trans* **by** *metis+*

Subterms

fun *subterms-term* :: 'a pset-term \Rightarrow 'a pset-term set **where**

$subterms\text{-term } (\emptyset n) = \{\emptyset n\}$
 $| subterms\text{-term } (Var i) = \{Var i\}$
 $| subterms\text{-term } (t1 \sqcup_s t2) = \{t1 \sqcup_s t2\} \cup subterms\text{-term } t1 \cup subterms\text{-term } t2$
 $| subterms\text{-term } (t1 \sqcap_s t2) = \{t1 \sqcap_s t2\} \cup subterms\text{-term } t1 \cup subterms\text{-term } t2$
 $| subterms\text{-term } (t1 -_s t2) = \{t1 -_s t2\} \cup subterms\text{-term } t1 \cup subterms\text{-term } t2$
 $| subterms\text{-term } (Single t) = \{Single t\} \cup subterms\text{-term } t$

fun *subterms-atom* :: 'a pset-atom \Rightarrow 'a pset-term set **where**

$subterms\text{-atom } (t1 \in_s t2) = subterms\text{-term } t1 \cup subterms\text{-term } t2$
 $| subterms\text{-atom } (t1 =_s t2) = subterms\text{-term } t1 \cup subterms\text{-term } t2$

definition *subterms-fm* :: 'a pset-fm \Rightarrow 'a pset-term set **where**
subterms-fm $\varphi \equiv \bigcup (\text{subterms-atom } \text{' atoms } \varphi)$

definition *subterms-branch* :: 'a branch \Rightarrow 'a pset-term set **where**
subterms-branch $b \equiv \bigcup (\text{subterms-fm } \text{' set } b)$

consts *subterms* :: 'a \Rightarrow 'b set

adhoc-overloading

subterms \Rightarrow *subterms-term* **and**

subterms \Rightarrow *subterms-atom* **and**

subterms \Rightarrow *subterms-fm* **and**

subterms \Rightarrow *subterms-branch*

lemma *subterms-fm-simps*[simp]:

subterms (Atom a) = *subterms* a

subterms (And p q) = *subterms* $p \cup$ *subterms* q

subterms (Or p q) = *subterms* $p \cup$ *subterms* q

subterms (Neg p) = *subterms* p

unfolding *subterms-fm-def* **by** *auto*

lemma *subterms-branch-simps*:

subterms [] = {}

subterms ($x \#$ xs) = *subterms* $x \cup$ *subterms* xs

unfolding *subterms-branch-def* **by** *auto*

lemma *subterms-refl*[simp]:

$t \in$ *subterms* t

by (*induction* t) *auto*

lemma *subterms-term-subterms-term-trans*:

$s \in$ *subterms-term* $t \Longrightarrow t \in$ *subterms-term* $v \Longrightarrow s \in$ *subterms-term* v

apply(*induction* v)

apply(*auto*)

done

lemma *subterms-term-subterms-atom-trans*:

$s \in$ *subterms-term* $t \Longrightarrow t \in$ *subterms-atom* $v \Longrightarrow s \in$ *subterms-atom* v

apply(*cases* v *rule: subterms-atom.cases*)

using *subterms-term-subterms-term-trans* **by** *auto*

lemma *subterms-term-subterms-fm-trans*:

$s \in$ *subterms-term* $t \Longrightarrow t \in$ *subterms-fm* $\varphi \Longrightarrow s \in$ *subterms-fm* φ

apply(*induction* φ)

apply(*auto simp: subterms-term-subterms-atom-trans*)

done

lemma *subterms-fmD*:

$t1 \sqcup_s t2 \in$ *subterms-fm* $\varphi \Longrightarrow t1 \in$ *subterms-fm* φ

$t1 \sqcup_s t2 \in$ *subterms-fm* $\varphi \Longrightarrow t2 \in$ *subterms-fm* φ

$t1 \sqcap_s t2 \in \text{subterms-fm } \varphi \implies t1 \in \text{subterms-fm } \varphi$
 $t1 \sqcap_s t2 \in \text{subterms-fm } \varphi \implies t2 \in \text{subterms-fm } \varphi$
 $t1 -_s t2 \in \text{subterms-fm } \varphi \implies t1 \in \text{subterms-fm } \varphi$
 $t1 -_s t2 \in \text{subterms-fm } \varphi \implies t2 \in \text{subterms-fm } \varphi$
Single $t \in \text{subterms-fm } \varphi \implies t \in \text{subterms-fm } \varphi$
by (*metis UnCI subterms-term.simps subterms-refl subterms-term-subterms-fm-trans*)⁺

lemma *subterms-branchD*:

$t1 \sqcup_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 \sqcup_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
 $t1 \sqcap_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 \sqcap_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
 $t1 -_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 -_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
Single $t \in \text{subterms-branch } b \implies t \in \text{subterms-branch } b$
unfolding *subterms-branch-def* **using** *subterms-fmD* **by** *fast*⁺

lemma *subterms-term-subterms-branch-trans*:

$s \in \text{subterms-term } t \implies t \in \text{subterms-branch } b \implies s \in \text{subterms-branch } b$
unfolding *subterms-branch-def* **using** *subterms-term-subterms-fm-trans* **by** *blast*

lemma *AT-mem-subterms-branchD*:

assumes *AT* $(s \in_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
using *assms* **unfolding** *subterms-branch-def* **by** *force*⁺

lemma *AF-mem-subterms-branchD*:

assumes *AF* $(s \in_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
using *assms* **unfolding** *subterms-branch-def* **by** *force*⁺

lemma *AT-eq-subterms-branchD*:

assumes *AT* $(s =_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
using *assms* **unfolding** *subterms-branch-def* **by** *force*⁺

lemma *AF-eq-subterms-branchD*:

assumes *AF* $(s =_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
using *assms* **unfolding** *subterms-branch-def* **by** *force*⁺

Interactions between Subterms and Subformulae

lemma *Un-vars-term-subterms-term-eq-vars-term*:

$\bigcup (\text{vars-term } ' \text{subterms } t) = \text{vars-term } t$
by (*induction t*) *auto*

lemma *Un-vars-term-subterms-fm-eq-vars-fm*:

$\bigcup (\text{vars-term } ' \text{subterms-fm } \varphi) = \text{vars-fm } \varphi$

proof(*induction* φ)
case (*Atom* a)
then show ?*case*
 by (*cases* a) (*auto simp: Un-vars-term-subterms-term-eq-vars-term*)
qed (*fastforce*)+

lemma *Un-vars-term-subterms-branch-eq-vars-branch*:
 $\bigcup(\text{vars-term } ' \text{ subterms-branch } b) = \text{vars-branch } b$
using *Un-vars-term-subterms-fm-eq-vars-fm*
unfolding *vars-branch-def subterms-branch-def*
by *force*

lemma *subs-vars-branch-if-sub-subterms-branch*:
 $\text{subterms-branch } b1 \subseteq \text{subterms-branch } b2 \implies \text{vars-branch } b1 \subseteq \text{vars-branch } b2$
using *Un-vars-term-subterms-branch-eq-vars-branch*
by (*metis complete-lattice-class.Sup-subset-mono subset-image-iff*)

lemma *subterms-branch-eq-if-vars-branch-eq*:
 $\text{subterms-branch } b1 = \text{subterms-branch } b2 \implies \text{vars-branch } b1 = \text{vars-branch } b2$
using *subs-vars-branch-if-sub-subterms-branch* **by** *blast*

lemma *mem-vars-term-if-mem-subterms-term*:
 $x \in \text{vars-term } s \implies s \in \text{subterms-term } t \implies x \in \text{vars-term } t$
apply(*induction* t)
 apply(*auto intro: pset-term.set-intros*)
done

lemma *mem-vars-fm-if-mem-subterms-fm*:
 $x \in \text{vars-term } s \implies s \in \text{subterms-fm } \varphi \implies x \in \text{vars-fm } \varphi$
proof(*induction* φ)
case (*Atom* a)
then show ?*case*
 by (*cases* a) (*auto simp: mem-vars-term-if-mem-subterms-term*)
qed (*auto simp: vars-fm-def*)

lemma *vars-term-sub-subterms-term*:
 $v \in \text{vars-term } t \implies \text{Var } v \in \text{subterms-term } t$
apply(*induction* t)
 apply(*auto*)
done

lemma *vars-atom-sub-subterms-atom*:
 $v \in \text{vars-atom } a \implies \text{Var } v \in \text{subterms-atom } a$
apply(*cases* a)
 apply(*auto simp: vars-term-sub-subterms-term*)
done

lemma *vars-fm-sub-subterms-fm*:

$v \in \text{vars-fm } \varphi \implies \text{Var } v \in \text{subterms-fm } \varphi$
apply(*induction* φ)
apply(*auto simp: vars-atom-subs-subterms-atom*)
done

lemma *vars-branch-subs-subterms-branch*:
 $\text{Var } v \in \text{vars-branch } b \subseteq \text{subterms-branch } b$
unfolding *vars-branch-def subterms-branch-def*
apply(*auto simp: vars-fm-subs-subterms-fm*)
done

lemma *subterms-term-subterms-atom-Atom-trans*:
 $\text{Atom } a \in \text{set } b \implies x \in \text{subterms-term } s \implies s \in \text{subterms-atom } a \implies x \in \text{subterms-branch } b$
unfolding *subterms-branch-def*
by (*metis UN-I subterms-fm-simps(1) subterms-term-subterms-atom-trans*)

lemma *subterms-branch-subterms-subterms-fm-trans*:
 $b \neq [] \implies x \in \text{subterms-term } t \implies t \in \text{subterms-fm } (\text{last } b) \implies x \in \text{subterms-branch } b$
using *subterms-branch-def subterms-term-subterms-fm-trans* **by** *fastforce*

Set Atoms in a Branch

abbreviation *pset-atoms-branch* :: $'a \text{ fm list} \Rightarrow 'a \text{ set}$ **where**
 $\text{pset-atoms-branch } b \equiv \bigcup (\text{atoms } ` \text{set } b)$

1.2.4 Finiteness of Variables, Subterms, and Subformulae

lemma *finite-vars-term*: $\text{finite } (\text{vars-term } t)$
apply(*induction* t)
apply(*auto*)
done

lemma *finite-vars-atom*: $\text{finite } (\text{vars-atom } a)$
apply(*cases* a)
apply(*auto simp: finite-vars-term*)
done

lemma *finite-vars-fm*: $\text{finite } (\text{vars-fm } \varphi)$
apply(*induction* φ)
apply(*auto simp: finite-vars-atom*)
done

lemma *finite-vars-branch*: $\text{finite } (\text{vars-branch } b)$
apply(*induction* b)
apply(*auto simp: vars-branch-def finite-vars-fm*)
done

lemma *finite-subterms-term*: $\text{finite } (\text{subterms-term } l)$

```

apply(induction l)
  apply(auto)
done

```

```

lemma finite-subterms-atom: finite (subterms-atom l)
  apply(cases l rule: subterms-atom.cases)
  apply(auto simp: finite-subterms-term)
done

```

```

lemma finite-subterms-fm: finite (subterms-fm  $\varphi$ )
  apply(induction  $\varphi$ )
  apply(auto simp: finite-subterms-atom)
done

```

```

lemma finite-subterms-branch: finite (subterms-branch b)
  apply(induction b)
  apply(auto simp: subterms-branch-def finite-subterms-fm)
done

```

```

lemma finite-subfms: finite (subfms  $\varphi$ )
  apply(induction  $\varphi$ )
  apply(auto)
done

```

```

lemma finite-subfms-branch: finite (subfms-branch b)
  by (induction b) (auto simp: subfms-branch-simps finite-subfms)

```

```

lemma finite-atoms: finite (atoms  $\varphi$ )
  by (induction  $\varphi$ ) auto

```

```

lemma finite-pset-atoms-branch: finite (pset-atoms-branch b)
  by (auto simp: finite-atoms)

```

1.2.5 Non-Emptiness of Subterms

```

lemma subterms-term-nonempty[simp]: subterms-term t  $\neq$  {}
  by (induction t) auto

```

```

lemma subterms-atom-nonempty[simp]: subterms-atom l  $\neq$  {}
  by (cases l rule: subterms-atom.cases) auto

```

```

lemma subterms-fm-nonempty[simp]: subterms-fm  $\varphi \neq$  {}
  by (induction  $\varphi$ ) auto

```

Chapter 2

A Tableau Calculus for MLSS

In this chapter, we define a tableau calculus for MLSS. Since we want this calculus to be compatible with Isabelle/HOL instead of a set theory, we introduce a typing system with which we can define the notion of urelements, i.e. elements that are not sets.

2.1 Typing Rules

We define the typing rules for set terms and atoms, as well as for formulae

inductive *types-pset-term* :: ('a \Rightarrow nat) \Rightarrow 'a pset-term \Rightarrow nat \Rightarrow bool ($\langle \vdash \ - \ - \ \vdash \ \rangle$ [46, 46, 46] 46) **where**

$v \vdash \emptyset \ n : \text{Suc } n$
| $v \vdash \text{Var } x : v \ x$
| $v \vdash t : l \Longrightarrow v \vdash \text{Single } t : \text{Suc } l$
| $v \vdash s : l \Longrightarrow v \vdash t : l \Longrightarrow l \neq 0 \Longrightarrow v \vdash s \sqcup_s t : l$
| $v \vdash s : l \Longrightarrow v \vdash t : l \Longrightarrow l \neq 0 \Longrightarrow v \vdash s \sqcap_s t : l$
| $v \vdash s : l \Longrightarrow v \vdash t : l \Longrightarrow l \neq 0 \Longrightarrow v \vdash s -_s t : l$

inductive-cases *types-pset-term-cases*:

$v \vdash \emptyset \ n : l \ v \vdash \text{Var } x : l \ v \vdash \text{Single } t : l$
 $v \vdash s \sqcup_s t : l \ v \vdash s \sqcap_s t : l \ v \vdash s -_s t : l$

lemma *types-pset-term-intros'*:

$l = \text{Suc } n \Longrightarrow v \vdash \emptyset \ n : l$
 $l = v \ x \Longrightarrow v \vdash \text{Var } x : l$
 $l \neq 0 \Longrightarrow v \vdash t : \text{nat.pred } l \Longrightarrow v \vdash \text{Single } t : l$
by (*auto simp add: types-pset-term.intros(1,2,3) pred-def split: nat.splits*)

definition *type-of-term* :: ('a \Rightarrow nat) \Rightarrow 'a pset-term \Rightarrow nat **where**

type-of-term $v \ t \equiv \text{THE } l. v \vdash t : l$

inductive *types-pset-atom* :: ('a ⇒ nat) ⇒ 'a pset-atom ⇒ bool **where**
 [[v ⊢ s : l; v ⊢ t : l]] ⇒ *types-pset-atom* v (s =_s t)
 | [[v ⊢ s : l; v ⊢ t : Suc l]] ⇒ *types-pset-atom* v (s ∈_s t)

definition *types-pset-fm* :: ('a ⇒ nat) ⇒ 'a pset-fm ⇒ bool **where**
types-pset-fm v φ ≡ (∀ a ∈ atoms φ. *types-pset-atom* v a)

consts *types* :: ('a ⇒ nat) ⇒ 'b ⇒ bool (**infix** ‹‹› 45)
adhoc-overloading *types* ⇒ *types-pset-atom* *types-pset-fm*

inductive-cases *types-pset-atom-Member-cases*:
 v ⊢ s ∈_s t1 ⊔_s t2 v ⊢ s ∈_s t1 ⊓_s t2 v ⊢ s ∈_s t1 -_s t2 v ⊢ s ∈_s Single t
abbreviation *urelem'* v (φ :: 'a pset-fm) t ≡ v ⊢ φ ∧ v ⊢ t : 0
definition *urelem* :: 'a pset-fm ⇒ 'a pset-term ⇒ bool **where**
urelem φ t ≡ (∃ v. *urelem'* v φ t)

2.2 The Calculus

We define a tableau calculus for MLSS build up from linear and branching expansion rules.

2.2.1 Closedness

fun *member-seq* :: 'a pset-term ⇒ 'a pset-atom list ⇒ 'a pset-term ⇒ bool **where**
member-seq s [] t ⇔ s = t
 | *member-seq* s ((s' ∈_s u) # cs) t ⇔ s = s' ∧ *member-seq* u cs t
 | *member-seq* - - - ⇔ False

fun *member-cycle* :: 'a pset-atom list ⇒ bool **where**
member-cycle ((s ∈_s t) # cs) = *member-seq* s ((s ∈_s t) # cs) s
 | *member-cycle* - = False

inductive *bclosed* :: 'a branch ⇒ bool **where**
contr: [[φ ∈ set b; Neg φ ∈ set b]] ⇒ *bclosed* b
 | *memEmpty*: AT (t ∈_s (∅ n)) ∈ set b ⇒ *bclosed* b
 | *neqSelf*: AF (t =_s t) ∈ set b ⇒ *bclosed* b
 | *memberCycle*: [[*member-cycle* cs; set cs ⊆ Atoms (set b)]] ⇒ *bclosed* b

abbreviation *bopen* b ≡ ¬ *bclosed* b

2.2.2 Linear Expansion Rules

fun *tlvl-terms* :: 'a pset-atom ⇒ 'a pset-term set **where**
tlvl-terms (t1 ∈_s t2) = {t1, t2}
 | *tlvl-terms* (t1 =_s t2) = {t1, t2}

lemma *tlvl-intros*[*intro*, *simp*]:

$t1 \in \text{tlvl-terms } (t1 \in_s t2)$
 $t2 \in \text{tlvl-terms } (t2 \in_s t1)$
 $t1 \in \text{tlvl-terms } (t1 =_s t2)$
 $t2 \in \text{tlvl-terms } (t2 =_s t1)$
by auto

fun *subst-tlvl* :: 'a pset-term \Rightarrow 'a pset-term \Rightarrow 'a pset-atom \Rightarrow 'a pset-atom **where**
subst-tlvl *t1 t2* ($s1 \in_s s2$) =
 (if $s1 = t1$ then $t2$ else $s1$) \in_s (if $s2 = t1$ then $t2$ else $s2$)
| *subst-tlvl* *t1 t2* ($s1 =_s s2$) =
 (if $s1 = t1$ then $t2$ else $s1$) $=_s$ (if $s2 = t1$ then $t2$ else $s2$)

inductive *lexpands-fm* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**
And $p\ q \in \text{set } b \Longrightarrow \text{lexpands-fm } [p, q]\ b$
| *Neg* ($Or\ p\ q \in \text{set } b \Longrightarrow \text{lexpands-fm } [Neg\ p, Neg\ q]\ b$)
| $[[\ Or\ p\ q \in \text{set } b; Neg\ p \in \text{set } b] \Longrightarrow \text{lexpands-fm } [q]\ b]$
| $[[\ Or\ p\ q \in \text{set } b; Neg\ q \in \text{set } b] \Longrightarrow \text{lexpands-fm } [p]\ b]$
| $[[\ Neg\ (And\ p\ q) \in \text{set } b; p \in \text{set } b] \Longrightarrow \text{lexpands-fm } [Neg\ q]\ b]$
| $[[\ Neg\ (And\ p\ q) \in \text{set } b; q \in \text{set } b] \Longrightarrow \text{lexpands-fm } [Neg\ p]\ b]$
| *Neg* ($Neg\ p \in \text{set } b \Longrightarrow \text{lexpands-fm } [p]\ b$)

inductive *lexpands-un* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**
AF ($s \in_s t1 \sqcup_s t2 \in \text{set } b \Longrightarrow \text{lexpands-un } [AF\ (s \in_s t1), AF\ (s \in_s t2)]\ b$)
| $[[\ AT\ (s \in_s t1) \in \text{set } b; t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-un } [AT\ (s \in_s t1 \sqcup_s t2)]\ b]$
| $[[\ AT\ (s \in_s t2) \in \text{set } b; t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-un } [AT\ (s \in_s t1 \sqcup_s t2)]\ b]$
| $[[\ AT\ (s \in_s t1 \sqcup_s t2) \in \text{set } b; AF\ (s \in_s t1) \in \text{set } b] \Longrightarrow \text{lexpands-un } [AT\ (s \in_s t2)]\ b]$
| $[[\ AT\ (s \in_s t1 \sqcup_s t2) \in \text{set } b; AF\ (s \in_s t2) \in \text{set } b] \Longrightarrow \text{lexpands-un } [AT\ (s \in_s t1)]\ b]$
| $[[\ AF\ (s \in_s t1) \in \text{set } b; AF\ (s \in_s t2) \in \text{set } b; t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-un } [AF\ (s \in_s t1 \sqcup_s t2)]\ b]$

inductive *lexpands-int* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**
AT ($s \in_s t1 \sqcap_s t2 \in \text{set } b \Longrightarrow \text{lexpands-int } [AT\ (s \in_s t1), AT\ (s \in_s t2)]\ b$)
| $[[\ AF\ (s \in_s t1) \in \text{set } b; t1 \sqcap_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-int } [AF\ (s \in_s t1 \sqcap_s t2)]\ b]$
| $[[\ AF\ (s \in_s t2) \in \text{set } b; t1 \sqcap_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-int } [AF\ (s \in_s t1 \sqcap_s t2)]\ b]$
| $[[\ AF\ (s \in_s t1 \sqcap_s t2) \in \text{set } b; AT\ (s \in_s t1) \in \text{set } b] \Longrightarrow \text{lexpands-int } [AF\ (s \in_s t2)]\ b]$
| $[[\ AF\ (s \in_s t1 \sqcap_s t2) \in \text{set } b; AT\ (s \in_s t2) \in \text{set } b] \Longrightarrow \text{lexpands-int } [AF\ (s \in_s t1)]\ b]$
| $[[\ AT\ (s \in_s t1) \in \text{set } b; AT\ (s \in_s t2) \in \text{set } b; t1 \sqcap_s t2 \in \text{subterms } (\text{last } b)] \Longrightarrow \text{lexpands-int } [AT\ (s \in_s t1 \sqcap_s t2)]\ b]$

inductive *lexpands-diff* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**
AT ($s \in_s t1 -_s t2 \in \text{set } b \Longrightarrow \text{lexpands-diff } [AT\ (s \in_s t1), AF\ (s \in_s t2)]\ b$)

$\llbracket AF (s \in_s t1) \in set\ b; t1 \text{ --}_s t2 \in subterms\ (last\ b) \rrbracket$
 $\implies lexpands\text{-}diff\ [AF (s \in_s t1 \text{ --}_s t2)]\ b$
 $\llbracket AT (s \in_s t2) \in set\ b; t1 \text{ --}_s t2 \in subterms\ (last\ b) \rrbracket$
 $\implies lexpands\text{-}diff\ [AF (s \in_s t1 \text{ --}_s t2)]\ b$
 $\llbracket AF (s \in_s t1 \text{ --}_s t2) \in set\ b; AT (s \in_s t1) \in set\ b \rrbracket$
 $\implies lexpands\text{-}diff\ [AT (s \in_s t2)]\ b$
 $\llbracket AF (s \in_s t1 \text{ --}_s t2) \in set\ b; AF (s \in_s t2) \in set\ b \rrbracket$
 $\implies lexpands\text{-}diff\ [AF (s \in_s t1)]\ b$
 $\llbracket AT (s \in_s t1) \in set\ b; AF (s \in_s t2) \in set\ b; t1 \text{ --}_s t2 \in subterms\ (last\ b) \rrbracket$
 $\implies lexpands\text{-}diff\ [AT (s \in_s t1 \text{ --}_s t2)]\ b$

inductive *lexpands-single* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**

Single $t1 \in subterms\ (last\ b) \implies lexpands\text{-}single\ [AT (t1 \in_s Single\ t1)]\ b$
 $\llbracket AT (s \in_s Single\ t1) \in set\ b \rrbracket \implies lexpands\text{-}single\ [AT (s =_s t1)]\ b$
 $\llbracket AF (s \in_s Single\ t1) \in set\ b \rrbracket \implies lexpands\text{-}single\ [AF (s =_s t1)]\ b$

inductive *lexpands-eq* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**

$\llbracket AT (t1 =_s t2) \in set\ b; AT\ l \in set\ b; t1 \in tlvl\text{-}terms\ l \rrbracket$
 $\implies lexpands\text{-}eq\ [AT (subst\text{-}tlvl\ t1\ t2\ l)]\ b$
 $\llbracket AT (t1 =_s t2) \in set\ b; AF\ l \in set\ b; t1 \in tlvl\text{-}terms\ l \rrbracket$
 $\implies lexpands\text{-}eq\ [AF (subst\text{-}tlvl\ t1\ t2\ l)]\ b$
 $\llbracket AT (t1 =_s t2) \in set\ b; AT\ l \in set\ b; t2 \in tlvl\text{-}terms\ l \rrbracket$
 $\implies lexpands\text{-}eq\ [AT (subst\text{-}tlvl\ t2\ t1\ l)]\ b$
 $\llbracket AT (t1 =_s t2) \in set\ b; AF\ l \in set\ b; t2 \in tlvl\text{-}terms\ l \rrbracket$
 $\implies lexpands\text{-}eq\ [AF (subst\text{-}tlvl\ t2\ t1\ l)]\ b$
 $\llbracket AT (s \in_s t) \in set\ b; AF (s' \in_s t) \in set\ b \rrbracket$
 $\implies lexpands\text{-}eq\ [AF (s =_s s')]\ b$

fun *polarise* :: bool \Rightarrow 'a fm \Rightarrow 'a fm **where**

polarise True $\varphi = \varphi$
polarise False $\varphi = Neg\ \varphi$

lemma *lexpands-eq-induct'*[consumes 1, case-names subst neg]:

assumes *lexpands-eq* $b'\ b$
assumes $\bigwedge t1\ t2\ t1'\ t2'\ p\ l\ b.$
 $\llbracket AT (t1 =_s t2) \in set\ b; polarise\ p (Atom\ l) \in set\ b;$
 $(t1',\ t2') \in \{(t1,\ t2), (t2,\ t1)\}; t1' \in tlvl\text{-}terms\ l \rrbracket$
 $\implies P [polarise\ p (Atom (subst\text{-}tlvl\ t1'\ t2'\ l))]\ b$
assumes $\bigwedge s\ t\ s'\ b. \llbracket AT (s \in_s t) \in set\ b; AF (s' \in_s t) \in set\ b \rrbracket \implies P [AF (s$
 $=_s\ s')]\ b$
shows $P\ b'\ b$
using *assms*(1)
apply(*induction* rule: *lexpands-eq.induct*)
by (*metis* *assms*(2-) *insertI1* *insertI2* *polarise.simps*)**+**

inductive *lexpands* :: 'a branch \Rightarrow 'a branch \Rightarrow bool **where**

lexpands-fm $b'\ b \implies lexpands\ b'\ b$
 $\llbracket lexpands\text{-}un\ b'\ b \rrbracket \implies lexpands\ b'\ b$
 $\llbracket lexpands\text{-}int\ b'\ b \rrbracket \implies lexpands\ b'\ b$

| *lexpands-diff* $b' b \implies \text{lexpands } b' b$
| *lexpands-single* $b' b \implies \text{lexpands } b' b$
| *lexpands-eq* $b' b \implies \text{lexpands } b' b$

lemma *lexpands-induct*[*consumes 1*]:

assumes *lexpands* $b' b$

shows

($\bigwedge p q b. \text{And } p q \in \text{set } b \implies P [p, q] b \implies$
 $\bigwedge p q b. \text{Neg } (\text{Or } p q) \in \text{set } b \implies P [\text{Neg } p, \text{Neg } q] b \implies$
 $\bigwedge p q b. \text{Or } p q \in \text{set } b \implies \text{Neg } p \in \text{set } b \implies P [q] b \implies$
 $\bigwedge p q b. \text{Or } p q \in \text{set } b \implies \text{Neg } q \in \text{set } b \implies P [p] b \implies$
 $\bigwedge p q b. \text{Neg } (\text{And } p q) \in \text{set } b \implies p \in \text{set } b \implies P [\text{Neg } q] b \implies$
 $\bigwedge p q b. \text{Neg } (\text{And } p q) \in \text{set } b \implies q \in \text{set } b \implies P [\text{Neg } p] b \implies$
 $\bigwedge p b. \text{Neg } (\text{Neg } p) \in \text{set } b \implies P [p] b \implies$
 $\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 \sqcup_s t2) \in \text{set } b \implies P [\text{AF } (s \in_s t1), \text{AF } (s \in_s t2)] b$)
 \implies
 $(\bigwedge s t1 b t2. \text{AT } (s \in_s t1) \in \text{set } b \implies t1 \sqcup_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AT } (s \in_s t1 \sqcup_s t2)] b \implies$
 $(\bigwedge s t2 b t1. \text{AT } (s \in_s t2) \in \text{set } b \implies t1 \sqcup_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AT } (s \in_s t1 \sqcup_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AT } (s \in_s t1 \sqcup_s t2) \in \text{set } b \implies \text{AF } (s \in_s t1) \in \text{set } b \implies P [\text{AT } (s \in_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AT } (s \in_s t1 \sqcup_s t2) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies P [\text{AT } (s \in_s t1)] b \implies$
 $(\bigwedge s t1 b t2. \text{AF } (s \in_s t1) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies t1 \sqcup_s t2 \in$
 $\text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 \sqcup_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AT } (s \in_s t1 \sqcap_s t2) \in \text{set } b \implies P [\text{AT } (s \in_s t1), \text{AT } (s \in_s t2)] b$)
 \implies
 $(\bigwedge s t1 b t2. \text{AF } (s \in_s t1) \in \text{set } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 \sqcap_s t2)] b \implies$
 $(\bigwedge s t2 b t1. \text{AF } (s \in_s t2) \in \text{set } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 \sqcap_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 \sqcap_s t2) \in \text{set } b \implies \text{AT } (s \in_s t1) \in \text{set } b \implies P [\text{AF } (s \in_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 \sqcap_s t2) \in \text{set } b \implies \text{AT } (s \in_s t2) \in \text{set } b \implies P [\text{AF } (s \in_s t1)] b \implies$
 $(\bigwedge s t1 b t2. \text{AT } (s \in_s t1) \in \text{set } b \implies \text{AT } (s \in_s t2) \in \text{set } b \implies t1 \sqcap_s t2 \in$
 $\text{subterms } (\text{last } b) \implies P [\text{AT } (s \in_s t1 \sqcap_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AT } (s \in_s t1 -_s t2) \in \text{set } b \implies P [\text{AT } (s \in_s t1), \text{AF } (s \in_s t2)] b$)
 \implies
 $(\bigwedge s t1 b t2. \text{AF } (s \in_s t1) \in \text{set } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 -_s t2)] b \implies$
 $(\bigwedge s t2 b t1. \text{AT } (s \in_s t2) \in \text{set } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 -_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 -_s t2) \in \text{set } b \implies \text{AT } (s \in_s t1) \in \text{set } b \implies P [\text{AT } (s \in_s t2)] b \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 -_s t2) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies P [\text{AF } (s \in_s t1)] b \implies$
 $(\bigwedge s t1 b t2. \text{AT } (s \in_s t1) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies t1 -_s t2 \in$

$subterms (last\ b) \implies P [AT (s \in_s t1 -_s t2)]\ b \implies$
 $(\bigwedge t1\ b. Single\ t1 \in subterms (last\ b) \implies P [AT (t1 \in_s Single\ t1)]\ b) \implies$
 $(\bigwedge s\ t1\ b. AT (s \in_s Single\ t1) \in set\ b \implies P [AT (s =_s t1)]\ b) \implies$
 $(\bigwedge s\ t1\ b. AF (s \in_s Single\ t1) \in set\ b \implies P [AF (s =_s t1)]\ b) \implies$
 $(\bigwedge t1\ t2\ b\ l. AT (t1 =_s t2) \in set\ b \implies AT\ l \in set\ b \implies t1 \in tlvl\text{-}terms\ l \implies$
 $P [AT (subst\text{-}tlvl\ t1\ t2\ l)]\ b) \implies$
 $(\bigwedge t1\ t2\ b\ l. AT (t1 =_s t2) \in set\ b \implies AF\ l \in set\ b \implies t1 \in tlvl\text{-}terms\ l \implies$
 $P [AF (subst\text{-}tlvl\ t1\ t2\ l)]\ b) \implies$
 $(\bigwedge t1\ t2\ b\ l. AT (t1 =_s t2) \in set\ b \implies AT\ l \in set\ b \implies t2 \in tlvl\text{-}terms\ l \implies$
 $P [AT (subst\text{-}tlvl\ t2\ t1\ l)]\ b) \implies$
 $(\bigwedge t1\ t2\ b\ l. AT (t1 =_s t2) \in set\ b \implies AF\ l \in set\ b \implies t2 \in tlvl\text{-}terms\ l \implies$
 $P [AF (subst\text{-}tlvl\ t2\ t1\ l)]\ b) \implies$
 $(\bigwedge s\ t\ b\ s'. AT (s \in_s t) \in set\ b \implies AF (s' \in_s t) \in set\ b \implies P [AF (s =_s s')]$
 $b) \implies P\ b'\ b$
using *assms*
apply(*induction rule: lexpands.induct*)
subgoal apply(*induction rule: lexpands-fm.induct*) **by** *metis+*
subgoal apply(*induction rule: lexpands-un.induct*) **by** *metis+*
subgoal apply(*induction rule: lexpands-int.induct*) **by** *metis+*
subgoal apply(*induction rule: lexpands-diff.induct*) **by** *metis+*
subgoal apply(*induction rule: lexpands-single.induct*) **by** *metis+*
subgoal apply(*induction rule: lexpands-eq.induct*) **by** *metis+*
done

2.2.3 Branching Expansion Rules

inductive *berpands-nowit* :: 'a branch set \Rightarrow 'a branch \Rightarrow bool **where**

$\llbracket Or\ p\ q \in set\ b;$
 $p \notin set\ b; Neg\ p \notin set\ b \rrbracket$
 $\implies\ berpands\text{-}nowit\ \{[p], [Neg\ p]\}\ b$
 $\mid \llbracket Neg\ (And\ p\ q) \in set\ b;$
 $Neg\ p \notin set\ b; p \notin set\ b \rrbracket$
 $\implies\ berpands\text{-}nowit\ \{[Neg\ p], [p]\}\ b$
 $\mid \llbracket AT (s \in_s t1 \sqcup_s t2) \in set\ b; t1 \sqcup_s t2 \in subterms (last\ b);$
 $AT (s \in_s t1) \notin set\ b; AF (s \in_s t1) \notin set\ b \rrbracket$
 $\implies\ berpands\text{-}nowit\ \{[AT (s \in_s t1)], [AF (s \in_s t1)]\}\ b$
 $\mid \llbracket AT (s \in_s t1) \in set\ b; t1 \sqcap_s t2 \in subterms (last\ b);$
 $AT (s \in_s t2) \notin set\ b; AF (s \in_s t2) \notin set\ b \rrbracket$
 $\implies\ berpands\text{-}nowit\ \{[AT (s \in_s t2)], [AF (s \in_s t2)]\}\ b$
 $\mid \llbracket AT (s \in_s t1) \in set\ b; t1 -_s t2 \in subterms (last\ b);$
 $AT (s \in_s t2) \notin set\ b; AF (s \in_s t2) \notin set\ b \rrbracket$
 $\implies\ berpands\text{-}nowit\ \{[AT (s \in_s t2)], [AF (s \in_s t2)]\}\ b$

inductive *berpands-wit* ::

'a pset-term \Rightarrow 'a pset-term \Rightarrow 'a \Rightarrow 'a branch set \Rightarrow 'a branch \Rightarrow bool **for** *t1 t2*
x where

$\llbracket AF (t1 =_s t2) \in set\ b; t1 \in subterms (last\ b); t2 \in subterms (last\ b);$
 $\nexists x. AT (x \in_s t1) \in set\ b \wedge AF (x \in_s t2) \in set\ b;$
 $\nexists x. AT (x \in_s t2) \in set\ b \wedge AF (x \in_s t1) \in set\ b;$

$$x \notin \text{vars } b; \neg \text{urelem } (\text{last } b) \ t1; \neg \text{urelem } (\text{last } b) \ t2 \] \\ \implies \text{bexpands-wit } t1 \ t2 \ x \ \{[AT \ (\text{Var } x \in_s \ t1), AF \ (\text{Var } x \in_s \ t2)], \\ [AT \ (\text{Var } x \in_s \ t2), AF \ (\text{Var } x \in_s \ t1)]\} \ b$$

inductive-cases *bexpands-wit-cases*[*consumes 1*]: *bexpands-wit* *t1 t2 x bs' b*

lemma *bexpands-witD*:

assumes *bexpands-wit* *t1 t2 x bs' b*

shows $bs' = \{[AT \ (\text{Var } x \in_s \ t1), AF \ (\text{Var } x \in_s \ t2)], \\ [AT \ (\text{Var } x \in_s \ t2), AF \ (\text{Var } x \in_s \ t1)]\}$

$AF \ (t1 =_s \ t2) \in \text{set } b \ t1 \in \text{subterms } (\text{last } b) \ t2 \in \text{subterms } (\text{last } b)$

$\nexists x. AT \ (x \in_s \ t1) \in \text{set } b \wedge AF \ (x \in_s \ t2) \in \text{set } b$

$\nexists x. AT \ (x \in_s \ t2) \in \text{set } b \wedge AF \ (x \in_s \ t1) \in \text{set } b$

$\neg \text{urelem } (\text{last } b) \ t1 \neg \text{urelem } (\text{last } b) \ t2$

$x \notin \text{vars } b$

using *bexpands-wit.cases*[*OF assms*] **by** *metis+*

inductive *bexpands* :: '*a branch set* \Rightarrow '*a branch* \Rightarrow *bool* **where**

bexpands-nowit *bs' b* \implies *bexpands* *bs' b*

| *bexpands-wit* *t1 t2 x bs' b* \implies *bexpands* *bs' b*

lemma *bexpands-disjnt*:

assumes *bexpands* *bs' b b' \in bs'*

shows $\text{set } b \cap \text{set } b' = \{\}$

using *assms*

proof(*induction* *bs' b* *rule: bexpands.induct*)

case (1 *bs b*)

then show *?case*

by (*induction rule: bexpands-nowit.induct*) (*auto intro: list.set-intros(1)*)

next

case (2 *t1 t2 x bs b*)

then show *?case*

proof(*induction rule: bexpands-wit.induct*)

case (1 *b*)

from $\langle x \notin \text{vars } b \rangle$

have $AT \ (\text{Var } x \in_s \ t1) \notin \text{set } b \ AF \ (\text{Var } x \in_s \ t1) \notin \text{set } b$

unfolding *vars-branch-def* **by** *auto*

with 1 **show** *?case*

by (*auto intro: list.set-intros(1) simp: disjnt-iff vars-fm-vars-branchI*)

qed

qed

lemma *bexpands-branch-not-Nil*:

assumes *bexpands* *bs' b b' \in bs'*

shows $b' \neq []$

using *assms*

proof(*induction* *bs' b* *rule: bexpands.induct*)

case (1 *bs' b*)

then show *?case*

```

  by (induction rule: bexpands-nowit.induct) auto
next
case (2 t1 t2 x bs' b)
then show ?case
  by (induction rule: bexpands-wit.induct) auto
qed

```

```

lemma bexpands-nonempty: bexpands bs' b  $\implies$  bs'  $\neq$  {}
proof(induction rule: bexpands.induct)
  case (1 bs' b)
  then show ?case by (induction rule: bexpands-nowit.induct) auto
next
  case (2 t1 t2 x bs' b)
  then show ?case by (induction rule: bexpands-wit.induct) auto
qed

```

```

lemma bexpands-strict-mono:
  assumes bexpands bs' b b'  $\in$  bs'
  shows set b  $\subset$  set (b' @ b)
  using bexpands-disjnt[OF assms] bexpands-branch-not-Nil[OF assms]
  by (simp add: less-le) (metis Un-Int-eq(1) set-empty2)

```

```

inductive-cases bexpands-cases[consumes 1, case-names no-param param]: bexpands bs b

```

2.2.4 Expansion Closure

```

inductive expandss :: 'a branch  $\Rightarrow$  'a branch  $\Rightarrow$  bool where
  expandss b b
| lexpands b3 b2  $\implies$  set b2  $\subset$  set (b3 @ b2)  $\implies$  expandss b2 b1  $\implies$  expandss (b3 @ b2) b1
| bexpands bs b2  $\implies$  b3  $\in$  bs  $\implies$  expandss b2 b1  $\implies$  expandss (b3 @ b2) b1

```

```

lemma expandss-trans: expandss b3 b2  $\implies$  expandss b2 b1  $\implies$  expandss b3 b1
  by (induction rule: expandss.induct) (auto simp: expandss.intros)

```

```

lemma expandss-suffix:
  expandss b1 b2  $\implies$  suffix b2 b1
  apply(induction rule: expandss.induct)
  apply(auto simp: suffix-appendI)
  done

```

```

lemmas expandss-mono = set-mono-suffix[OF expandss-suffix]

```

```

lemma expandss-last-eq[simp]:
  expandss b' b  $\implies$  b  $\neq$  []  $\implies$  last b' = last b
  by (metis expandss-suffix last-appendR suffix-def)

```

```

lemma expandss-not-Nil:

```

$expandss\ b'\ b \implies b \neq [] \implies b' \neq []$
using *expandss-suffix suffix-bot.extremum-uniqueI* **by** *blast*

2.2.5 Well-Formed Branch

definition *wf-branch* $b \equiv \exists \varphi. expandss\ b\ [\varphi]$

lemma *wf-branch-singleton[simp]*: *wf-branch* $[\varphi]$
unfolding *wf-branch-def* **using** *expandss.intros(1)* **by** *blast*

lemma *wf-branch-not-Nil[simp, intro?]*: *wf-branch* $b \implies b \neq []$
unfolding *wf-branch-def*
using *expandss-suffix suffix-bot.extremum-uniqueI* **by** *blast*

lemma *wf-branch-expandss*: *wf-branch* $b \implies expandss\ b'\ b \implies wf-branch\ b'$
using *expandss-trans wf-branch-def* **by** *blast*

lemma *wf-branch-lexpands*:
wf-branch $b \implies lexpands\ b'\ b \implies set\ b \subset set\ (b' @ b) \implies wf-branch\ (b' @ b)$
by (*metis expandss.simps wf-branch-expandss*)

Chapter 3

A Type Checker for MLSS

In this chapter, we define a type checker for MLSS that determines which variables of an input formula are urelements.

For this we reduce the type checking problem to the theory of the successor function.

3.1 Solver for the Theory of the Successor Function

We implement a solver for the theory consisting of variables, 0, and the successor function. We only deal with equality and not with disequality or inequality. Disequalities of the form $l \neq 0$ are translated to $l = \text{Suc } x$ for some fresh x .

Note that disequalities and inequalities can always be fulfilled by choosing large enough values for the variables.

```
datatype 'a suc-term = Var 'a | Zero | Succ nat 'a suc-term
```

```
datatype 'a suc-atom = is-Eq: Eq 'a suc-term 'a suc-term | is-NEq: NEq 'a suc-term 'a suc-term
```

```
lemma finite-set-suc-term[simp]: finite (set-suc-term t)  
  by (induction t) auto
```

```
lemma finite-set-suc-atom[simp]: finite (set-suc-atom a)  
  by (cases a) auto
```

```
fun succ :: nat  $\Rightarrow$  'a suc-term  $\Rightarrow$  'a suc-term where  
  succ n (Succ k t) = succ (n + k) t  
| succ 0 t = t  
| succ n t = Succ n t
```

```
fun is-Succ-normal-term :: 'a suc-term  $\Rightarrow$  bool where
```

```

  is-Succ-normal-term (Var -)  $\longleftrightarrow$  True
| is-Succ-normal-term Zero  $\longleftrightarrow$  True
| is-Succ-normal-term (Succ n Zero)  $\longleftrightarrow$   $n \neq 0$ 
| is-Succ-normal-term (Succ n (Var -))  $\longleftrightarrow$   $n \neq 0$ 
| is-Succ-normal-term (Succ - (Succ - -))  $\longleftrightarrow$  False

```

lemma *not-is-Succ-normal-Succ-0[simp]*: \neg *is-Succ-normal-term* (Succ 0 t)
by (cases t) auto

lemma *is-Succ-normal-term-SuccD[simp]*: *is-Succ-normal-term* (Succ n t) \implies *is-Succ-normal-term* t
by (cases t) auto

fun *is-Succ-normal-atom* :: 'a suc-atom \Rightarrow bool **where**
is-Succ-normal-atom (Eq t1 t2) \longleftrightarrow *is-Succ-normal-term* t1 \wedge *is-Succ-normal-term* t2
| *is-Succ-normal-atom* (NEq t1 t2) \longleftrightarrow *is-Succ-normal-term* t1 \wedge *is-Succ-normal-term* t2

consts *is-Succ-normal* :: 'a \Rightarrow bool

adhoc-overloading *is-Succ-normal* \equiv *is-Succ-normal-term is-Succ-normal-atom*

fun *I-term* :: ('a \Rightarrow nat) \Rightarrow 'a suc-term \Rightarrow nat **where**
I-term v (Var x) = v x
| *I-term* v Zero = 0
| *I-term* v (Succ n t) = (Suc $\hat{\sim}$ n) (*I-term* v t)

fun *I-atom* :: ('a \Rightarrow nat) \Rightarrow 'a suc-atom \Rightarrow bool **where**
I-atom v (Eq t1 t2) \longleftrightarrow *I-term* v t1 = *I-term* v t2
| *I-atom* v (NEq t1 t2) \longleftrightarrow *I-term* v t1 \neq *I-term* v t2

fun *subst-term* :: ('a \Rightarrow 'a suc-term) \Rightarrow 'a suc-term \Rightarrow 'a suc-term **where**
subst-term σ (Var x) = succ 0 (σ x)
| *subst-term* - Zero = Zero
| *subst-term* σ (Succ n t) = succ n (*subst-term* σ t)

fun *subst-atom* :: ('a \Rightarrow 'a suc-term) \Rightarrow 'a suc-atom \Rightarrow 'a suc-atom **where**
subst-atom σ (Eq t1 t2) = Eq (*subst-term* σ t1) (*subst-term* σ t2)
| *subst-atom* σ (NEq t1 t2) = NEq (*subst-term* σ t1) (*subst-term* σ t2)

lemma *I-term-succ*: *I-term* v (succ n t) = *I-term* v (Succ n t)
by (induction n t rule: succ.induct) auto

lemma *is-Succ-normal-succ[simp]*: *is-Succ-normal* (succ n t)
by (induction n t rule: succ.induct) auto

lemma *is-Succ-normal-subst-term[simp]*: *is-Succ-normal* (*subst-term* σ t)
by (induction t) auto

lemma *is-Succ-normal-subst-atom*[simp]: *is-Succ-normal* (subst-atom σ a)
by (cases a) simp-all

lemma *is-NEq-subst-atom*[simp]:
is-NEq (subst-atom v a) \longleftrightarrow *is-NEq* a
by (cases a) auto

abbreviation *solve-Var-Eq-Succ* **where**
solve-Var-Eq-Succ solve x n t es \equiv
 (if (Var x) = t
 then (if $n = 0$ then solve es else None)
 else map-option ((#) (Eq (Var x) (succ n t)))
 (solve (map (subst-atom (Var(x := succ n t))) es))
)

lemma *size-succ-leq*[termination-simp]: size (succ n t) \leq Suc (size t)
by (induction n t rule: succ.induct) auto

function (sequential) *solve* :: 'a suc-atom list \Rightarrow 'a suc-atom list option **where**
solve [] = Some []
 | solve (Eq (Var x) (Var y) # es) = solve-Var-Eq-Succ solve x 0 (Var y) es
 | solve (Eq (Var x) (Succ n t) # es) = solve-Var-Eq-Succ solve x n t es
 | solve (Eq (Succ n t) (Var x) # es) = solve-Var-Eq-Succ solve x n t es
 | solve (Eq (Succ n s) (Succ k t) # es) =
 (if $n \geq k$
 then solve (Eq t (succ ($n - k$) s) # es)
 else solve (Eq s (succ ($k - n$) t) # es)
)
 | solve (Eq Zero Zero # es) = solve es
 | solve (Eq Zero (Var x) # es) = solve-Var-Eq-Succ solve x 0 Zero es
 | solve (Eq (Var x) Zero # es) = solve-Var-Eq-Succ solve x 0 Zero es
 | solve (Eq Zero (Succ 0 t) # es) = solve (Eq t Zero # es)
 | solve (Eq Zero (Succ n t) # es) = None
 | solve (Eq (Succ 0 t) Zero # es) = solve (Eq t Zero # es)
 | solve (Eq (Succ n t) Zero # es) = None
by pat-completeness auto
termination by size-change

abbreviation *is-normal* $a \equiv \neg$ *is-NEq* $a \wedge$ *is-Succ-normal* a

lemma *is-Succ-normal-solve*:
assumes solve es = Some $ss \forall a \in$ set es . *is-normal* a
assumes $a \in$ set ss
shows *is-Succ-normal* a
using *assms*
by (induction es arbitrary: ss rule: solve.induct) (auto split: if-splits)

lemma *I-term-subst-term*:

assumes $I\text{-atom } v (Eq (Var\ x) t)$
shows $I\text{-term } v (subst\text{-term } (Var(x := t)) s) = I\text{-term } v s$
using *assms*
by (*induction s*) (*auto simp: I-term-succ*)

lemma *I-atom-subst-atom*:
assumes $I\text{-atom } v (Eq (Var\ x) t)$
shows $I\text{-atom } v (subst\text{-atom } (Var(x := t)) a) \longleftrightarrow I\text{-atom } v a$
using *assms*
by (*cases a*) (*auto simp: I-term-subst-term*)

lemma *I-atom-solve-None*:
assumes $solve\ es = None \ \forall a \in set\ es.\ is\text{-normal } a$
shows $\exists a \in set\ es.\ \neg I\text{-atom } v a$
proof –
have *False* **if** $\forall a \in set\ es.\ I\text{-atom } v a$
using *assms that*
by (*induction es rule: solve.induct*)
(force simp: I-atom-subst-atom I-term-succ split: if-splits)+
then show *?thesis*
by *blast*
qed

lemma *set-suc-term-succ[simp]*: $set\text{-suc-term } (succ\ n\ t) = set\text{-suc-term } t$
by (*induction n t rule: succ.induct*) *auto*

lemma *not-mem-subst-term-self*:
assumes $x \notin set\text{-suc-term } t$
shows $x \notin set\text{-suc-term } (subst\text{-term } (Var(x := t)) s)$
using *assms*
by (*induction s*) *auto*

lemma *not-mem-subst-atom-self*:
assumes $x \notin set\text{-suc-term } t$
shows $x \notin set\text{-suc-atom } (subst\text{-atom } (Var(x := t)) a)$
using *not-mem-subst-term-self[OF assms]* **by** (*cases a*) *simp-all*

lemma *not-mem-subst-term*:
assumes $z \notin set\text{-suc-term } t \ z \notin set\text{-suc-term } s$
shows $z \notin set\text{-suc-term } (subst\text{-term } (Var(x := t)) s)$
using *assms*
by (*induction s*) *auto*

lemma *not-mem-subst-atom*:
assumes $z \notin set\text{-suc-term } t \ z \notin set\text{-suc-atom } a$
shows $z \notin set\text{-suc-atom } (subst\text{-atom } (Var(x := t)) a)$
using *not-mem-subst-term[OF assms(1)] assms(2)* **by** (*cases a*) *simp-all*

lemma *not-mem-suc-atom-solve*:

```

assumes solve es = Some ss  $\forall a \in \text{set es. is-normal } a$ 
assumes  $\forall a \in \text{set es. } z \notin \text{set-suc-atom } a$ 
shows  $\forall a \in \text{set ss. } z \notin \text{set-suc-atom } a$ 
using assms
by (induction es arbitrary: ss rule: solve.induct)
    (force simp: not-mem-subst-atom split: if-splits)+

lemma not-mem-suc-atom-if-solve:
assumes solve es = Some (Eq (Var x) t # ss)  $\forall a \in \text{set es. is-normal } a$ 
shows  $\forall a \in \text{set ss. } x \notin \text{set-suc-atom } a$ 
using assms
proof(induction es arbitrary: ss rule: solve.induct)
  case (2 y z es)
    note not-mem-suc-atom-solve[where ?es=map (subst-atom (Var(y := Var z)))
    es and ?z=y]
    with 2 show ?case
      by (auto simp: not-mem-subst-atom-self split: if-splits)
  next
    case (3 y n t es)
    note not-mem-suc-atom-solve[
      where ?es=map (subst-atom (Var(y := succ n t))) es and ?z=y]
    with 3 show ?case
      apply (simp split: if-splits)
      by (metis is-Succ-normal-term.simps(5) not-mem-subst-atom-self set-suc-term-succ
      suc-term.set-cases)
  next
    case (4 n t y es)
    note not-mem-suc-atom-solve[
      where ?es=map (subst-atom (Var(y := succ n t))) es and ?z=y]
    with 4 show ?case
      apply (simp split: if-splits)
      by (metis is-Succ-normal-term.simps(5) not-mem-subst-atom-self set-suc-term-succ
      suc-term.set-cases)
  next
    case (7 y es)
    note not-mem-suc-atom-solve[
      where ?es=map (subst-atom (Var(y := Zero))) es and ?z=y]
    with 7 show ?case
      by (auto simp: not-mem-subst-atom-self split: if-splits)
  next
    case (8 y es)
    note not-mem-suc-atom-solve[
      where ?es=map (subst-atom (Var(y := Zero))) es and ?z=y]
    with 8 show ?case
      by (auto simp: not-mem-subst-atom-self split: if-splits)
qed (auto split: if-splits)

fun assign :: 'a suc-atom list  $\Rightarrow$  ('a  $\Rightarrow$  nat) where
  assign [] = ( $\lambda x. 0$ )

```

| $assign (Eq (Var x) (Var y) \# ss) = (let\ ass = assign\ ss\ in\ ass(x := ass\ y))$
| $assign (Eq (Var x) (Succ\ n\ (Var\ y)) \# ss) = (let\ ass = assign\ ss\ in\ ass(x := (Succ\ \widetilde{n})\ (ass\ y)))$
| $assign (Eq (Var x) Zero \# ss) = (let\ ass = assign\ ss\ in\ ass(x := 0))$
| $assign (Eq (Var x) (Succ\ n\ Zero) \# ss) = (let\ ass = assign\ ss\ in\ ass(x := (Succ\ \widetilde{n})\ 0))$
| $assign (Eq (Var x) (Succ\ n\ (Succ\ k\ t)) \# ss) = assign (Eq (Var x) (Succ\ (n + k)\ t) \# ss)$

lemma *assign-succ*:

$assign (Eq (Var x) (succ\ n\ t) \# ss) = assign (Eq (Var x) (Succ\ n\ t) \# ss)$
by (*induction n t rule: succ.induct*) *auto*

lemma *I-term-fun-upd*:

assumes $x \notin set\ suc\ term\ t$
shows $I\ term\ (v(x := s))\ t = I\ term\ v\ t$
using *assms* **by** (*induction t*) *auto*

lemma *I-atom-fun-upd*:

assumes $x \notin set\ suc\ atom\ a$
shows $I\ atom\ (v(x := s))\ a = I\ atom\ v\ a$
using *assms* **by** (*cases a*) (*auto simp: I-term-fun-upd*)

lemma *I-atom-fun-updI*:

assumes $x \notin set\ suc\ atom\ a$ $I\ atom\ v\ a$
shows $I\ atom\ (v(x := s))\ a$
using *assms* *I-atom-fun-upd* **by** *metis*

lemma *I-atom-assign-if-solve-Some*:

assumes $solve\ es = Some\ ss \ \forall a \in set\ es.\ is\ normal\ a$
shows $\forall a \in set\ ss.\ I\ atom\ (assign\ ss)\ a$
using *assms*

proof(*induction es arbitrary: ss rule: solve.induct*)

case ($2\ x\ y\ es$)

note *not-mem-suc-atom-if-solve*[**where** $?es = Eq (Var\ x) (Var\ y) \# es$ **and** $?x = x$]

with 2 **show** $?case$

by (*auto simp: Let-def I-atom-fun-upd split: if-splits*)

next

case ($3\ x\ n\ t\ es$)

note *not-mem-suc-atom-if-solve*[**where** $?es = Eq (Var\ x) (Succ\ n\ t) \# es$ **and** $?x = x$]

with 3 **show** $?case$

by (*cases t*)

(*auto simp: Let-def I-term-succ I-atom-fun-upd assign-succ split: if-splits*)

next

case ($4\ n\ t\ x\ es$)

note *not-mem-suc-atom-if-solve*[**where** $?es = Eq (Var\ x) (Succ\ n\ t) \# es$ **and** $?x = x$]

with 4 **show** $?case$

```

    by (cases t)
      (auto simp: Let-def I-term-succ I-atom-fun-upd assign-succ split: if-splits)
next
case (7 x es)
note not-mem-suc-atom-if-solve[where ?es=Eq (Var x) Zero # es and ?x=x]
with 7 show ?case
  by (auto simp: Let-def I-atom-fun-upd split: if-splits)
next
case (8 x es)
note not-mem-suc-atom-if-solve[where ?es=Eq (Var x) Zero # es and ?x=x]
with 8 show ?case
  by (auto simp: Let-def I-atom-fun-upd split: if-splits)
qed (auto split: if-splits)

```

lemma *I-atom-iff-if-I-atom-solve-Some:*
assumes $\text{solve } es = \text{Some } ss \ \forall a \in \text{set } es. \text{ is-normal } a$
shows $(\forall a \in \text{set } ss. \text{ I-atom } v \ a) \longleftrightarrow (\forall a \in \text{set } es. \text{ I-atom } v \ a)$
using *assms*
by (*induction es arbitrary: ss rule: solve.induct*)
 (auto simp: I-term-succ I-atom-subst-atom split: if-splits)

lemma *assign-minimal-if-solve-Some:*
assumes $\text{solve } es = \text{Some } ss \ \forall a \in \text{set } es. \text{ is-normal } a$
assumes $\forall a \in \text{set } ss. \text{ I-atom } v \ a$
shows $\text{assign } ss \ z \leq v \ z$
using *assms*
proof(*induction es arbitrary: ss z rule: solve.induct*)
 case (2 x y es)
 note not-mem-suc-atom-if-solve[where ?es=Eq (Var x) (Var y) # es and ?x=x]
 with 2 show ?case
 by (auto simp: Let-def split: if-splits)
next
 case (3 x n t es)
 note not-mem-suc-atom-if-solve[where ?es=Eq (Var x) (Succ n t) # es and
 ?x=x]
 with 3 show ?case
 by (cases t) (auto simp: Let-def I-term-succ assign-succ split: if-splits)
next
 case (4 n t x es)
 note not-mem-suc-atom-if-solve[where ?es=Eq (Var x) (Succ n t) # es and
 ?x=x]
 with 4 show ?case
 by (cases t) (auto simp: Let-def I-term-succ assign-succ split: if-splits)
qed (auto split: if-splits)

fun *elim-NEq-Zero-aux* :: ('a::fresh) *set* \Rightarrow 'a *suc-atom list* \Rightarrow 'a *suc-atom list*
where
elim-NEq-Zero-aux - [] = []
 | *elim-NEq-Zero-aux* *us* (NEq (Var x) Zero # *es*) =

(let $fx = \text{fresh us } x \text{ in } Eq (Var x) (Succ 1 (Var fx)) \# \text{elim-NEq-Zero-aux (insert } fx \text{ us) es}$)

| $\text{elim-NEq-Zero-aux us (e \# es) = e \# elim-NEq-Zero-aux us es}$

definition $\text{elim-NEq-Zero} :: ('a::\text{fresh}) \text{ suc-atom list} \Rightarrow 'a \text{ suc-atom list}$
where $\text{elim-NEq-Zero es} \equiv \text{elim-NEq-Zero-aux } (\bigcup (\text{set-suc-atom ' set es})) \text{ es}$

lemma $\text{is-normal-elim-NEq-Zero-aux}$:

assumes $\forall a \in \text{set es. is-Eq } a \longrightarrow \text{is-normal } a$

assumes $\forall a \in \text{set es. is-NEq } a \longrightarrow (\exists x. a = \text{NEq (Var } x) \text{ Zero})$

shows $\forall a \in \text{set (elim-NEq-Zero-aux us es). is-normal } a$

using assms

by ($\text{induction es rule: elim-NEq-Zero-aux.induct}$) ($\text{auto simp: Let-def}$)

lemma $\text{is-normal-elim-NEq-Zero}$:

assumes $\forall a \in \text{set es. is-Eq } a \longrightarrow \text{is-normal } a$

assumes $\forall a \in \text{set es. is-NEq } a \longrightarrow (\exists x. a = \text{NEq (Var } x) \text{ Zero})$

shows $\forall a \in \text{set (elim-NEq-Zero es). is-normal } a$

using $\text{is-normal-elim-NEq-Zero-aux[OF assms]}$ **unfolding** elim-NEq-Zero-def **by** blast

lemma $\text{I-atom-Var-NEq-Zero-if-I-atom-Var-Eq-Succ}$:

$\text{I-atom } v (Eq (Var x) (Succ 1 (Var fx))) \Longrightarrow \text{I-atom } v (\text{NEq (Var } x) \text{ Zero})$

by simp

lemma $\text{I-atom-if-I-atom-elim-NEq-Zero-aux}$:

assumes $\forall a \in \text{set (elim-NEq-Zero-aux us es). I-atom } v a$

shows $\forall a \in \text{set es. I-atom } v a$

using $\text{assms I-atom-Var-NEq-Zero-if-I-atom-Var-Eq-Succ}$

by ($\text{induction us es rule: elim-NEq-Zero-aux.induct}$) ($\text{auto simp: Let-def}$)

lemma $\text{I-atom-if-I-atom-elim-NEq-Zero}$:

assumes $\forall a \in \text{set (elim-NEq-Zero es). I-atom } v a$

shows $\forall a \in \text{set es. I-atom } v a$

using $\text{assms I-atom-if-I-atom-elim-NEq-Zero-aux}$

unfolding elim-NEq-Zero-def **by** blast

lemma $\text{I-term-if-eq-on-set-suc-term}$:

assumes $\forall x \in \text{set-suc-term } t. v' x = v x$

shows $\text{I-term } v' t = \text{I-term } v t$

using assms

by ($\text{induction } t$) auto

lemma $\text{I-atom-if-eq-on-set-suc-atom}$:

assumes $\forall x \in \text{set-suc-atom } a. v' x = v x$

shows $\text{I-atom } v' a = \text{I-atom } v a$

using assms

by ($\text{cases } a$) ($\text{simp;metis I-term-if-eq-on-set-suc-term UnI1 UnI2}$) $+$

lemma *not-mem-set-suc-atom-elim-NEq-zero-aux*:
assumes $finite\ us \cup (set\ suc\ atom\ ' \ set\ es) \subseteq us$
assumes $a \in set\ (elim\ NEq\ Zero\ aux\ us\ es)$
assumes $x \in us - \cup (set\ suc\ atom\ ' \ set\ es)$
shows $x \notin set\ suc\ atom\ a$
using *assms*
by (*induction us es arbitrary; x rule: elim-NEq-Zero-aux.induct*)
(auto simp: fresh-notIn Let-def)

lemma *I-atom-elim-NEq-Zero-aux-if-I-atom*:
assumes $\cup (set\ suc\ atom\ ' \ set\ es) \subseteq us\ finite\ us$
assumes $\forall a \in set\ es.\ I\ atom\ v\ a$
obtains v' **where** $\forall a \in set\ (elim\ NEq\ Zero\ aux\ us\ es).\ I\ atom\ v'\ a$
 $\forall x \in us.\ v'\ x = v\ x$
using *assms*
proof(*induction us es arbitrary; thesis rule: elim-NEq-Zero-aux.induct*)
case (1 *us*)
then show *?case by auto*
next
case (2 *us x es thesis*)
then obtain v' **where**
 $v': \forall a \in set\ (elim\ NEq\ Zero\ aux\ (insert\ (fresh\ us\ x)\ us)\ es).\ I\ atom\ v'\ a$
 $\forall x \in insert\ (fresh\ us\ x)\ us.\ v'\ x = v\ x$
by (*auto simp: subset-insertI2*)
define v'' **where** $v'' \equiv v'(fresh\ us\ x := nat.pred\ (v'\ x))$

from $v'\ 2.prem\ 3$ **have** $fresh\ us\ x \notin us \ \forall x \in us.\ v''\ x = v\ x$
unfolding $v''\text{-def}$ **using** *fresh-notIn* **by** (*metis fun-upd-apply insertCI*)
moreover from $2.prem\ 2$ **have** $x \in us$
by *auto*
moreover from $2.prem\ 4$ **have** $v\ x \neq 0$
by *simp*
with $v'\ \langle x \in us \rangle$ **have** $v'\ x \neq 0$
by *simp*
with $\langle x \in us \rangle$ **have** $v\ x = Suc\ (v''\ (fresh\ us\ x))$
unfolding $v''\text{-def}$ **by** (*auto simp: v'(2)*)
moreover have $I\ atom\ v''\ a = I\ atom\ v'\ a$
if $a \in set\ (elim\ NEq\ Zero\ aux\ (insert\ (fresh\ us\ x)\ us)\ es)$ **for** a
proof –
have $fresh\ us\ x \in insert\ (fresh\ us\ x)\ us - \cup (set\ suc\ atom\ ' \ set\ es)$
using $2.prem\ 2$ $\langle fresh\ us\ x \notin us \rangle$ **by** *auto*
note *not-mem-set-suc-atom-elim-NEq-zero-aux[OF - - - this]*
with that have $fresh\ us\ x \notin set\ suc\ atom\ a$
using $2.prem\ 2,3$ **by** *auto*
then show *?thesis*
unfolding $v''\text{-def}$ **by** (*simp add: I-atom-fun-upd*)

qed
ultimately show *?case*
by (*intro 2(2)[where ?v'=v'']*) (*auto simp: v'(1) Let-def*)

qed (*simp*; *metis I-term-if-eq-on-set-suc-term in-mono*)+

lemma *I-atom-elim-NEq-Zero-if-I-atom*:

assumes $\forall a \in \text{set } es. I\text{-atom } v \ a$

obtains v' **where** $\forall a \in \text{set } (elim\text{-NEq-Zero } es). I\text{-atom } v' \ a$
 $\forall x \in \bigcup (set\text{-suc-atom } ' \text{ set } es). v' \ x = v \ x$

proof –

have *finite* ($\bigcup (set\text{-suc-atom } ' \text{ set } es)$)

by *simp*

note *I-atom-elim-NEq-Zero-aux-if-I-atom*[*OF* - *this assms*]

with *that show* *?thesis*

unfolding *elim-NEq-Zero-def* **by** *blast*

qed

lemma *not-I-atom-if-solve-elim-NEq-Zero-None*:

assumes $\forall a \in \text{set } es. is\text{-Eq } a \longrightarrow is\text{-normal } a$

assumes $\forall a \in \text{set } es. is\text{-NEq } a \longrightarrow (\exists x. a = NEq (Var \ x) \ Zero)$

assumes *solve* (*elim-NEq-Zero es*) = *None*

shows $\exists a \in \text{set } es. \neg I\text{-atom } v \ a$

proof –

from *is-normal-elim-NEq-Zero assms* **have** $\forall a \in \text{set } (elim\text{-NEq-Zero } es). is\text{-normal } a$

by *blast*

note *I-atom-solve-None*[*OF assms*(3) *this*]

then **have** $\exists a \in \text{set } (elim\text{-NEq-Zero } es). \neg I\text{-atom } v \ a$ **for** v

by *blast*

with *I-atom-elim-NEq-Zero-if-I-atom* **show** *?thesis*

by *metis*

qed

lemma

assumes $\forall a \in \text{set } es. is\text{-Eq } a \longrightarrow is\text{-normal } a$

assumes $\forall a \in \text{set } es. is\text{-NEq } a \longrightarrow (\exists x. a = NEq (Var \ x) \ Zero)$

assumes *solve* (*elim-NEq-Zero es*) = *Some ss*

shows *I-atom-assign-if-solve-elim-NEq-Zero-Some*:

$\forall a \in \text{set } es. I\text{-atom } (assign \ ss) \ a$

and *I-atom-assign-minimal-if-solve-elim-NEq-Zero-Some*:

$\llbracket \forall a \in \text{set } es. I\text{-atom } v \ a; z \in \bigcup (set\text{-suc-atom } ' \text{ set } es) \rrbracket$

$\implies assign \ ss \ z \leq v \ z$

proof –

from *is-normal-elim-NEq-Zero assms* **have** *normal*: $\forall a \in \text{set } (elim\text{-NEq-Zero } es). is\text{-normal } a$

by *blast*

note *I-atom-assign-if-solve-Some*[*OF assms*(3) *normal*]

note *this*[*unfolded I-atom-iff-if-I-atom-solve-Some*[*OF assms*(3) *normal*]]

from *I-atom-if-I-atom-elim-NEq-Zero*[*OF this*] **show** $\forall a \in \text{set } es. I\text{-atom } (assign \ ss) \ a$.

note *assign-minimal-if-solve-Some*[*OF assms*(3) *normal*]

then have *assign ss* $z \leq v z$ **if** $\forall a \in \text{set } (\text{elim-NEq-Zero } es)$. *I-atom v a* **for** *v*
using *that I-atom-iff-if-I-atom-solve-Some[OF assms(3) normal]* **by** *blast*
then show *assign ss* $z \leq v z$
if $\forall a \in \text{set } es$. *I-atom v a* $z \in \bigcup (\text{set-suc-atom } \text{' set } es)$ **for** *v*
using *that I-atom-elim-NEq-Zero-if-I-atom* **by** *metis*
qed

theory *MLSS-Typing*
imports *MLSS-Calculus*
begin

3.2 Typing and Branch Expansion

We prove that the branch expansion rules preserve well-typedness.

lemma *types-term-unique*:
shows $v \vdash t : l1 \implies v \vdash t : l2 \implies l2 = l1$
apply(*induction arbitrary: l2 rule: types-pset-term.induct*)
apply (*metis types-pset-term-cases*)
done

lemma *type-of-term-if-types-term*:
shows $v \vdash t : l \implies \text{type-of-term } v t = l$
using *types-term-unique unfolding type-of-term-def* **by** *blast*

lemma *types-term-if-mem-subterms-term*:
assumes $s \in \text{subterms } t$
assumes $v \vdash t : lt$
shows $\exists ls. v \vdash s : ls$
using *assms*
by (*induction t arbitrary: s lt*) (*auto elim: types-pset-term-cases*)

lemma *is-Var-if-types-term-0*:
shows $v \vdash t : 0 \implies \text{is-Var } t$
by (*induction t*) (*auto elim: types-pset-term-cases*)

lemma *is-Var-if-urelem'*: $\text{urelem}' v \varphi t \implies \text{is-Var } t$
using *is-Var-if-types-term-0* **by** *blast*

lemma *is-Var-if-urelem*: $\text{urelem } \varphi t \implies \text{is-Var } t$
unfolding *urelem-def* **using** *is-Var-if-urelem'* **by** *blast*

lemma *types-fmD*:
 $v \vdash \text{And } p q \implies v \vdash p$
 $v \vdash \text{And } p q \implies v \vdash q$
 $v \vdash \text{Or } p q \implies v \vdash p$
 $v \vdash \text{Or } p q \implies v \vdash q$
 $v \vdash \text{Neg } p \implies v \vdash p$

$v \vdash \text{Atom } a \implies v \vdash a$
unfolding *types-pset-fm-def* **using** *fm.set-intros* **by** *auto*

lemma *types-fmI*:

$v \vdash p \implies v \vdash q \implies v \vdash \text{And } p \ q$
 $v \vdash p \implies v \vdash q \implies v \vdash \text{Or } p \ q$
 $v \vdash p \implies v \vdash \text{Neg } p$
 $v \vdash a \implies v \vdash \text{Atom } a$

unfolding *types-pset-fm-def* **using** *fm.set-intros* **by** *auto*

lemma *types-pset-atom-Member-D*:

includes *no member-ASCII-syntax*
assumes $v \vdash s \in_s f \ t1 \ t2 \ f \in \{(\sqcup_s), (\sqcap_s), (-_s)\}$
shows $v \vdash s \in_s t1 \ v \vdash s \in_s t2$

proof –

from *assms* **obtain** *ls* **where**

$v \vdash s : ls \ v \vdash f \ t1 \ t2 : \text{Suc } ls$

using *types-pset-atom.simps* **by** *fastforce*

with *assms* **have** $v \vdash s \in_s t1 \wedge v \vdash s \in_s t2$

by (*auto simp: types-pset-atom.simps elim: types-pset-term-cases*)

then show $v \vdash s \in_s t1 \ v \vdash s \in_s t2$

by *blast+*

qed

lemmas *types-pset-atom-Member-Union-D = types-pset-atom-Member-D*[**where** $?f=(\sqcup_s)$, *simplified*]

and *types-pset-atom-Member-Inter-D = types-pset-atom-Member-D*[**where** $?f=(\sqcap_s)$, *simplified*]

and *types-pset-atom-Member-Diff-D = types-pset-atom-Member-D*[**where** $?f=(-_s)$, *simplified*]

lemma *types-term-if-mem-subterms*:

includes *no member-ASCII-syntax*

fixes $\varphi :: 'a \ \text{pset-fm}$

assumes $v \vdash \varphi$

assumes $f \ t1 \ t2 \in \text{subterms } \varphi \ f \in \{(\sqcup_s), (\sqcap_s), (-_s)\}$

obtains *lt* **where** $v \vdash t1 : lt \ v \vdash t2 : lt$

proof –

from *assms* **obtain** $a :: 'a \ \text{pset-atom}$ **where** *atom*: $v \vdash a \ f \ t1 \ t2 \in \text{subterms } a$

unfolding *types-pset-fm-def* **by** (*induction* φ) *auto*

obtain $t' \ l$ **where** $v \vdash t' : l \ f \ t1 \ t2 \in \text{subterms } t'$

apply(*rule types-pset-atom.cases*[*OF* $\langle v \vdash a \rangle$])

using *atom(2)* **by** *auto*

then obtain *lt* **where** $v \vdash t1 : lt \wedge v \vdash t2 : lt$

by (*induction* t' *arbitrary*: *l*)

(*use assms(3)* **in** $\langle \text{auto elim: types-pset-term-cases} \rangle$)

with that show *?thesis*

by *blast*

qed

lemma *types-if-types-Member-and-subterms*:
fixes $\varphi :: 'a \text{ pset-fm}$
assumes $v \vdash s \in_s t1 \vee v \vdash s \in_s t2 \vee v \vdash \varphi$
assumes $f \ t1 \ t2 \in \text{subterms } \varphi \ f \in \{(\sqcup_s), (\sqcap_s), (-_s)\}$
shows $v \vdash s \in_s f \ t1 \ t2$
proof –
from *types-term-if-mem-subterms*[*OF assms(2-)*] **obtain** *lt* **where** *lt*:
 $v \vdash t1 : lt \vee v \vdash t2 : lt$
by *blast*
moreover from *assms(1) lt(1,2)* **obtain** *ls* **where** $v \vdash s : ls \ \text{lt} = \text{Suc } ls$
by (*auto simp: types-pset-atom.simps dest: types-term-unique*)
ultimately show *?thesis*
using *assms*
by (*auto simp: types-pset-term.intros types-pset-atom.simps*)
qed

lemma *types-subst-trlvl*:
includes *no member-ASCII-syntax*
fixes $l :: 'a \text{ pset-atom}$
assumes $v \vdash AT \ (t1 =_s t2) \ v \vdash l$
shows $v \vdash \text{subst-trlvl } t1 \ t2 \ l$
proof –
from *assms* **obtain** *lt* **where** $v \vdash t1 : lt \vee v \vdash t2 : lt$
by (*auto simp: types-pset-atom.simps dest!: types-fmD(6)*)
with *assms(2)* **show** *?thesis*
by (*cases (t1, t2, l) rule: subst-trlvl.cases*)
(*auto simp: types-pset-atom.simps dest: types-term-unique*)
qed

lemma *types-sym-Equal*:
assumes $v \vdash t1 =_s t2$
shows $v \vdash t2 =_s t1$
using *assms* **by** (*auto simp: types-pset-atom.simps*)

lemma *types-lexpands*:
fixes $\varphi :: 'a \text{ pset-fm}$
assumes *lexpands* $b' \ b \ b \neq [] \ \varphi \in \text{set } b'$
assumes $\bigwedge (\varphi :: 'a \text{ pset-fm}). \ \varphi \in \text{set } b \implies v \vdash \varphi$
shows $v \vdash \varphi$
using *assms*
proof(*induction rule: lexpands.induct*)
case (1 *b' b*)
then show *?case*
apply(*induction rule: lexpands-fm.induct*)
apply(*force dest: types-fmD intro: types-fmI(3)*)
done
next
case (2 *b' b*)

```

then show ?case
proof(induction rule: lexpands-un.induct)
  case (1 s t1 t2 b)
  then show ?thesis
    by (auto dest!: types-fmD(5,6) 1(4) intro!: types-fmI(2,3,4)
      intro: types-pset-atom-Member-Union-D)
next
  case (2 s t1 b t2)
  then have v ⊢ last b
    by auto
  from types-if-types-Member-and-subterms[OF - this] 2 show ?case
    by (auto dest!: 2(5) types-fmD(6) intro: types-fmI(4))
next
  case (3 s t2 b t1)
  then have v ⊢ last b
    by auto
  from types-if-types-Member-and-subterms[OF - this] 3 show ?case
    by (auto dest!: 3(5) types-fmD(6) intro: types-fmI(4))
next
  case (4 s t1 t2 b)
  then show ?case
    by (auto dest!: types-fmD(5,6) 4(5) intro!: types-fmI(2,3,4)
      intro: types-pset-atom-Member-Union-D)
next
  case (5 s t1 t2 b)
  then show ?case
    by (auto dest!: types-fmD(5,6) 5(5) intro!: types-fmI(2,3,4)
      intro: types-pset-atom-Member-Union-D)
next
  case (6 s t1 b t2)
  then have v ⊢ last b
    by auto
  note types-if-types-Member-and-subterms[where ?f=(⊔s), OF - this 6(3), sim-
plified]
  with 6 show ?case
    by (auto dest!: types-fmD(5,6) 6(6) intro!: types-fmI(2,3,4))
qed
next
  case (3 b' b)
  then show ?case
proof(induction rule: lexpands-int.induct)
  case (1 s t1 t2 b)
  then show ?thesis
    by (auto dest!: types-fmD(5,6) 1(4) intro!: types-fmI(2,3,4)
      intro: types-pset-atom-Member-Inter-D)
next
  case (2 s t1 b t2)
  then have v ⊢ last b
    by auto

```

```

    from types-if-types-Member-and-subterms[OF - this] 2 show ?case
      by (auto dest!: 2(5) types-fmD(5,6) intro!: types-fmI(3,4))
next
  case (3 s t2 b t1)
  then have v ⊢ last b
    by auto
  from types-if-types-Member-and-subterms[OF - this] 3 show ?case
    by (auto dest!: 3(5) types-fmD(5,6) intro!: types-fmI(3,4))
next
  case (4 s t1 t2 b)
  then show ?case
    by (auto dest!: types-fmD(5,6) 4(5) intro!: types-fmI(2,3,4)
        intro: types-pset-atom-Member-Inter-D)
next
  case (5 s t1 t2 b)
  then show ?case
    by (auto dest!: types-fmD(5,6) 5(5) intro!: types-fmI(2,3,4)
        intro: types-pset-atom-Member-Inter-D)
next
  case (6 s t1 b t2)
  then have v ⊢ last b
    by auto
  note types-if-types-Member-and-subterms[where ?f=( $\prod_s$ ), OF - this 6(3), sim-
  plified]
  with 6 show ?case
    by (auto dest!: types-fmD(5,6) 6(6) intro!: types-fmI(2,3,4))
qed
next
  case (4 b' b)
  then show ?case
    proof(induction rule: lexpands-diff.induct)
      case (1 s t1 t2 b)
      then show ?thesis
        by (auto dest!: types-fmD(5,6) 1(4) intro!: types-fmI(2,3,4)
            intro: types-pset-atom-Member-Diff-D)
    next
      case (2 s t1 b t2)
      then have v ⊢ last b
        by auto
      from types-if-types-Member-and-subterms[OF - this] 2 show ?case
        by (auto dest!: 2(5) types-fmD(5,6) intro!: types-fmI(3,4))
    next
      case (3 s t2 b t1)
      then have v ⊢ last b
        by auto
      from types-if-types-Member-and-subterms[OF - this] 3 show ?case
        by (auto dest!: 3(5) types-fmD(5,6) intro!: types-fmI(3,4))
    next
      case (4 s t1 t2 b)

```

```

then show ?case
  by (auto dest!: types-fmD(5,6) 4(5) intro!: types-fmI(2,3,4)
      intro: types-pset-atom-Member-Diff-D)
next
  case (5 s t1 t2 b)
  then show ?case
    by (auto dest!: types-fmD(5,6) 5(5) intro!: types-fmI(2,3,4)
        intro: types-pset-atom-Member-Diff-D)
next
  case (6 s t1 b t2)
  then have v ⊢ last b
    by auto
    note types-if-types-Member-and-subterms[where ?f=(-s), OF - this 6(3),
simplified]
    with 6 show ?case
      by (auto dest!: types-fmD(5,6) 6(6) intro!: types-fmI(2,3,4))
  qed
next
  case (5 b' b)
  then show ?case
  proof(cases rule: lexpands-single.cases)
    case (1 t1)
    with 5 have v ⊢ last b
      by auto
    with ⟨Single t1 ∈ subterms (last b)⟩ obtain a :: 'a pset-atom
      where atom: a ∈ atoms (last b) Single t1 ∈ subterms a v ⊢ a
      unfolding types-pset-fm-def by (metis UN-E subterms-fm-def)
    obtain t' l where Single t1 ∈ subterms t' v ⊢ t' : l
      apply(rule types-pset-atom.cases[OF ⟨v ⊢ a⟩])
      using atom(2) by auto
    note types-term-if-mem-subterms-term[OF this]
    then obtain lt1 where v ⊢ t1 : lt1 v ⊢ Single t1 : Suc lt1
      by (metis types-pset-term-cases(3))
    with 5 1 show ?thesis
      using types-pset-atom.intros(2) types-pset-fm-def by fastforce
  qed (auto simp: types-pset-atom.simps elim!: types-pset-term-cases
      dest!: types-fmD(5,6) 5(4) intro!: types-fmI(3,4))
next
  case (6 b' b)
  then show ?case
  proof(cases rule: lexpands-eq.cases)
    case (5 s t s')
    with 6 show ?thesis
      by (auto 0 3 dest!: 6(4) types-fmD(5,6) dest: types-term-unique
          simp: types-pset-atom.simps types-term-unique intro!: types-fmI(3,4))
  qed (auto simp: types-sym-Equal dest!: 6(4) types-fmD(5,6)
      intro!: types-fmI(3,4) types-subst-tlvl)
qed

```

```

lemma types-bexpands-nowit:
  fixes  $\varphi :: 'a \text{ pset-fm}$ 
  assumes bexpands-nowit  $bs' b b' \in bs' \varphi \in \text{set } b'$ 
  assumes  $\bigwedge(\varphi :: 'a \text{ pset-fm}). \varphi \in \text{set } b \implies v \vdash \varphi$ 
  shows  $v \vdash \varphi$ 
  using assms(1)
proof(cases rule: bexpands-nowit.cases)
  case (1 p q)
  from assms 1(2) show ?thesis
    unfolding 1(1)
    by (auto dest!: assms(4) types-fmD(3) intro!: types-fmI(3))
  next
  case (2 p q)
  from assms 2(2) show ?thesis
    unfolding 2(1)
    by (auto dest!: assms(4) types-fmD(5) dest: types-fmD(1,2) intro!: types-fmI(3))
  next
  case (3 s t1 t2)
  from assms 3(2,3) show ?thesis
    unfolding 3(1) using types-pset-atom-Member-Union-D(1)[of v s t1 t2]
    by (auto dest!: types-fmD(6) assms(4) intro!: types-fmI(3,4))
  next
  case (4 s t1 t2)
  with assms have  $v \vdash \text{last } b$ 
    by (metis empty-iff empty-set last-in-set)
  from assms 4(2,3) show ?thesis
    unfolding 4(1)
    using types-if-types-Member-and-subterms[where ?f=( $\sqcap_s$ ), OF -  $\langle v \vdash \text{last } b \rangle$ ]
    4(3),
    THEN types-pset-atom-Member-Inter-D(2)]
    by (force dest!: types-fmD(6) assms(4) intro!: types-fmI(3,4))
  next
  case (5 s t1 t2)
  with assms have  $v \vdash \text{last } b$ 
    by (metis empty-iff empty-set last-in-set)
  from assms 5(2,3) show ?thesis
    unfolding 5(1)
    using types-if-types-Member-and-subterms[where ?f=( $-_s$ ), OF -  $\langle v \vdash \text{last } b \rangle$ ]
    5(3),
    THEN types-pset-atom-Member-Diff-D(2)]
    by (force dest!: types-fmD(6) assms(4) intro!: types-fmI(3,4))
qed

lemma types-term-if-on-vars-eq:
  assumes  $\forall x \in \text{vars } t. v' x = v x$ 
  shows  $v' \vdash t : l \iff v \vdash t : l$ 
  using assms
  apply(induction t arbitrary: l)
    apply(auto intro!: types-pset-term-intros' types-pset-term.intros(4-))

```

```

      elim!: types-pset-term-cases)
done

lemma types-pset-atom-if-on-vars-eq:
  fixes a :: 'a pset-atom
  assumes  $\forall x \in \text{vars } a. v' x = v x$ 
  shows  $v' \vdash a \longleftrightarrow v \vdash a$ 
  using assms
  by (auto simp: ball-Un types-pset-atom.simps dest!: types-term-if-on-vars-eq)

lemma types-pset-fm-if-on-vars-eq:
  fixes  $\varphi :: 'a \text{ pset-fm}$ 
  assumes  $\forall x \in \text{vars } \varphi. v' x = v x$ 
  shows  $v' \vdash \varphi \longleftrightarrow v \vdash \varphi$ 
  using assms types-pset-atom-if-on-vars-eq
  unfolding types-pset-fm-def vars-fm-def by fastforce

lemma types-term-fun-upd:
  assumes  $x \notin \text{vars } t$ 
  shows  $v(x := l) \vdash t : l \longleftrightarrow v \vdash t : l$ 
  using assms types-term-if-on-vars-eq by (metis fun-upd-other)

lemma types-pset-atom-fun-upd:
  fixes a :: 'a pset-atom
  assumes  $x \notin \text{vars } a$ 
  shows  $v(x := l) \vdash a \longleftrightarrow v \vdash a$ 
  using assms types-pset-atom-if-on-vars-eq by (metis fun-upd-other)

lemma types-pset-fm-fun-upd:
  fixes  $\varphi :: 'a \text{ pset-fm}$ 
  assumes  $x \notin \text{vars } \varphi$ 
  shows  $v(x := l) \vdash \varphi \longleftrightarrow v \vdash \varphi$ 
  using assms types-pset-fm-if-on-vars-eq by (metis fun-upd-other)

lemma types-bexpands-wit:
  fixes b :: 'a branch and bs' :: 'a branch set
  assumes bexpands-wit t1 t2 x bs' b b  $\neq []$ 
  assumes  $\bigwedge (\varphi :: 'a \text{ pset-fm}). \varphi \in \text{set } b \implies v \vdash \varphi$ 
  obtains l where  $\forall \varphi \in \text{set } b. v(x := l) \vdash \varphi$ 
     $\forall b' \in \text{bs}'. \forall \varphi \in \text{set } b'. v(x := l) \vdash \varphi$ 
  using assms(1)
proof(cases rule: bexpands-wit.cases)
  case 1
  from assms(3)[OF 1(2)] obtain lt where  $lt: v \vdash t1 : lt \ v \vdash t2 : lt$ 
  by (auto dest!: types-fmD simp: types-pset-atom.simps)
  with 1 assms(2,3) have  $lt \neq 0$ 
  unfolding urelem-def using last-in-set by metis
  with lt obtain ltp where  $ltp: v \vdash t1 : \text{Suc } ltp \ v \vdash t2 : \text{Suc } ltp$ 
  using not0-implies-Suc by blast

```

with *assms*(3) **have** $\forall \varphi \in \text{set } b. v(x := \text{lt}) \vdash \varphi$
using *types-pset-fm-fun-upd* $\langle x \notin \text{vars } b \rangle$ **by** (*metis vars-fm-vars-branchI*)
moreover from $\langle x \notin \text{vars } b \rangle \langle AF (t1 =_s t2) \in \text{set } b \rangle$ **have** *not-in-vars*: $x \notin \text{vars } t1 \ x \notin \text{vars } t2$
using *assms*(2) **by** (*auto simp: vars-fm-vars-branchI*)
from *this*[*THEN types-term-fun-upd*] **have** $\forall b' \in \text{bs}'. \forall \varphi \in \text{set } b'. v(x := \text{lt}) \vdash \varphi$
using *lt* **unfolding** 1(1)
apply(*auto intro!*: *types-fmI types-pset-term-intros'*(2) *simp: types-pset-atom.simps*)
apply (*metis fun-upd-same fun-upd-upd types-pset-term.intros*(2))+
done
ultimately show ?*thesis*
using *that* **by** *blast*
qed

lemma *types-expandss*:
fixes $b \ b' :: 'a \text{ branch}$
assumes *expandss* $b' \ b \ b \neq []$
assumes $\bigwedge \varphi. \varphi \in \text{set } b \implies v \vdash \varphi$
obtains v' **where** $\forall x \in \text{vars } b. v' \ x = v \ x \ \forall \varphi \in \text{set } b'. v' \vdash \varphi$
using *assms*
proof(*induction* $b' \ b$ *arbitrary: thesis rule: expandss.induct*)
case (1 b)
then show ?*case* **by** *blast*
next
case (2 $b3 \ b2 \ b1$)
then obtain v' **where** $v': \forall x \in \text{vars } b1. v' \ x = v \ x \ \forall \varphi \in \text{set } b2. v' \vdash \varphi$
by *blast*
with *types-lexpands*[*OF* $\langle \text{lexpands } b3 \ b2 \rangle$] **have** $\forall \varphi \in \text{set } b3. v' \vdash \varphi$
using *expandss-not-Nil*[*OF* $\langle \text{expandss } b2 \ b1 \rangle \langle b1 \neq [] \rangle$] **by** *blast*
with $v' \ 2.\text{prems}$ **show** ?*case*
by *force*
next
case (3 $bs \ b2 \ b3 \ b1$)
then obtain v' **where** $v': \forall x \in \text{vars } b1. v' \ x = v \ x \ \forall \varphi \in \text{set } b2. v' \vdash \varphi$
by *blast*
from $\langle \text{bexpands } bs \ b2 \rangle$ **show** ?*case*
proof(*cases* *rule: bexpands.cases*)
case 1
from *types-bexpands-nowit*[*OF this*] $v' \ \langle b3 \in bs \rangle$ **have** $\forall \varphi \in \text{set } b3. v' \vdash \varphi$
by *blast*
with $v' \ 3.\text{prems}$ **show** ?*thesis*
by *force*
next
case (2 $t1 \ t2 \ x$)
from *types-bexpands-wit*[*OF this*] $v' \ \langle b3 \in bs \rangle$ **obtain** l
where $\forall \varphi \in \text{set } b3. v'(x := l) \vdash \varphi$
using *expandss-not-Nil*[*OF* $\langle \text{expandss } b2 \ b1 \rangle \langle b1 \neq [] \rangle$] **by** *metis*
moreover from *bexpands-witD*(9)[*OF* 2] **have** $x \notin \text{vars } b1$

using *expandss-mono*[*OF* $\langle \text{expandss } b2 \ b1 \rangle$] **unfolding** *vars-branch-def* **by**
blast
then have $\forall y \in \text{vars } b1. (v'(x := l)) \ y = v \ y$
using *v'(1)* **by** *simp*
moreover from $\langle x \notin \text{vars } b2 \rangle \ v'(2)$ **have** $\forall \varphi \in \text{set } b2. v'(x := l) \vdash \varphi$
by (*meson types-pset-fm-fun-upd vars-fm-vars-branchI*)
ultimately show *?thesis*
using *v' 3.premis(1)*[**where** $?v'=v'(x := l)$] **by** *fastforce*
qed
qed

lemma *urelem-invar-if-wf-branch*:

assumes *wf-branch* *b*
assumes *urelem* (*last b*) $x \in \text{subterms } (\text{last } b)$
shows $\exists v. \forall \varphi \in \text{set } b. \text{urelem}' \ v \ \varphi \ x$
proof –
from *assms* **obtain** *v* **where** $v: v \vdash \text{last } b \ v \vdash x : 0$
unfolding *urelem-def* **by** *blast*
moreover from *assms* **have** *expandss b [last b]*
by (*metis expandss-last-eq last.simps list.distinct(1) wf-branch-def*)
from *types-expandss*[*OF this, simplified*] **obtain** *v'* **where**
 $\forall x \in \text{vars } (\text{last } b). v' \ x = v \ x \ \forall \varphi \in \text{set } b. v' \vdash \varphi$
by (*metis list.set-intros(1) vars-fm-vars-branchI*)
ultimately show *?thesis*
unfolding *urelem-def* **using** *assms*
by (*metis mem-vars-fm-if-mem-subterms-fm types-term-if-on-vars-eq*)
qed

lemma *not-types-term-0-if-types-term*:

fixes *s* :: 'a *pset-term*
assumes $f \ t1 \ t2 \in \text{subterms } s \ f \in \{(\sqcap_s), (\sqcup_s), (-_s)\}$
assumes $v \vdash f \ t1 \ t2 : l$
shows $\neg v \vdash t1 : 0 \ \neg v \vdash t2 : 0$
using *assms*
by (*induction s arbitrary: l*)
(auto elim: types-pset-term-cases dest: types-term-unique)

lemma *types-term-subterms*:

assumes $t \in \text{subterms } s$
assumes $v \vdash s : ls$
obtains *lt* **where** $v \vdash t : lt$
using *assms*
by (*induction s arbitrary: ls*) (*auto elim: types-pset-term-cases dest: types-term-unique*)

lemma *types-atom-subterms*:

fixes *a* :: 'a *pset-atom*
assumes $t \in \text{subterms } a$
assumes $v \vdash a$
obtains *lt* **where** $v \vdash t : lt$

```

using assms
by (cases a) (fastforce elim: types-term-subterms simp: types-pset-atom.simps)+

lemma subterms-type-pset-fm-not-None:
  fixes  $\varphi :: 'a \text{ pset-fm}$ 
  assumes  $t \in \text{subterms } \varphi$ 
  assumes  $v \vdash \varphi$ 
  obtains  $lt$  where  $v \vdash t : lt$ 
  using assms
  by (induction  $\varphi$ ) (auto elim: types-atom-subterms dest: types-fmD(1-5) dest!: types-fmD(6))

lemma not-urelem-comps-if-compound:
  assumes  $f \ t1 \ t2 \in \text{subterms } \varphi \ f \in \{(\sqcap_s), (\sqcup_s), (-_s)\}$ 
  shows  $\neg \text{urelem } \varphi \ t1 \ \neg \text{urelem } \varphi \ t2$ 
proof -
  from assms have  $\neg v \vdash t1 : 0 \ \neg v \vdash t2 : 0$  if  $v \vdash \varphi$  for  $v$ 
  using that not-types-term-0-if-types-term[OF - - subterms-type-pset-fm-not-None]
  using subterms-refl by metis+
  then show  $\neg \text{urelem } \varphi \ t1 \ \neg \text{urelem } \varphi \ t2$ 
  unfolding urelem-def by blast+
qed

```

3.3 Typing the Urelements

We define a recursive procedure that generates typing constraints. We then prove that the constraints can be solved with *MLSS-Suc-Theory.assign*. The solution then gives us the urelements.

```

abbreviation (input) SVar  $\equiv \text{MLSS-Suc-Theory.Var}$ 
abbreviation (input) SEq  $\equiv \text{MLSS-Suc-Theory.Eq}$ 
abbreviation (input) SNEq  $\equiv \text{MLSS-Suc-Theory.NEq}$ 
abbreviation (input) ssolve  $\equiv \text{MLSS-Suc-Theory.solve}$ 
abbreviation (input) sassign  $\equiv \text{MLSS-Suc-Theory.assign}$ 

```

```

fun constrs-term :: ('a pset-term  $\Rightarrow$  'b)  $\Rightarrow$  'a pset-term  $\Rightarrow$  'b suc-atom list where
  constrs-term  $n$  (Var  $x$ ) = [SEq (SVar ( $n$  (Var  $x$ ))) (SVar ( $n$  (Var  $x$ )))]
| constrs-term  $n$  ( $\emptyset$   $k$ ) = [SEq (SVar ( $n$  ( $\emptyset$   $k$ ))) (Succ (Suc  $k$ ) Zero)]
| constrs-term  $n$  ( $t1 \sqcup_s t2$ ) =
  [SEq (SVar ( $n$  ( $t1 \sqcup_s t2$ ))) (SVar ( $n$   $t1$ )), SEq (SVar ( $n$   $t1$ )) (SVar ( $n$   $t2$ )),
SNEq (SVar ( $n$   $t1$ )) Zero]
  @ constrs-term  $n$   $t1$  @ constrs-term  $n$   $t2$ 
| constrs-term  $n$  ( $t1 \sqcap_s t2$ ) =
  [SEq (SVar ( $n$  ( $t1 \sqcap_s t2$ ))) (SVar ( $n$   $t1$ )), SEq (SVar ( $n$   $t1$ )) (SVar ( $n$   $t2$ )),
SNEq (SVar ( $n$   $t1$ )) Zero]
  @ constrs-term  $n$   $t1$  @ constrs-term  $n$   $t2$ 
| constrs-term  $n$  ( $t1 -_s t2$ ) =
  [SEq (SVar ( $n$  ( $t1 -_s t2$ ))) (SVar ( $n$   $t1$ )), SEq (SVar ( $n$   $t1$ )) (SVar ( $n$   $t2$ )),

```

$SNEq (SVar (n t1)) Zero]$
 $@ constrs-term n t1 @ constrs-term n t2$
 $| constrs-term n (Single t) =$
 $[SEq (SVar (n (Single t))) (Succ 1 (SVar (n t)))]$
 $@ constrs-term n t$

fun *constrs-atom* :: ('a pset-term \Rightarrow 'b) \Rightarrow 'a pset-atom \Rightarrow 'b suc-atom list **where**
 $constrs-atom n (t1 =_s t2) =$
 $[SEq (SVar (n t1)) (SVar (n t2))]$
 $@ constrs-term n t1 @ constrs-term n t2$
 $| constrs-atom n (t1 \in_s t2) =$
 $[SEq (SVar (n t2)) (Succ 1 (SVar (n t1)))]$
 $@ constrs-term n t1 @ constrs-term n t2$

fun *constrs-fm* :: ('a pset-term \Rightarrow 'b) \Rightarrow 'a pset-fm \Rightarrow 'b suc-atom list **where**
 $constrs-fm n (Atom a) = constrs-atom n a$
 $| constrs-fm n (And p q) = constrs-fm n p @ constrs-fm n q$
 $| constrs-fm n (Or p q) = constrs-fm n p @ constrs-fm n q$
 $| constrs-fm n (Neg p) = constrs-fm n p$

lemma *is-Succ-normal-constrs-term*:

$\forall a \in set (constrs-term n t). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
by (*induction t*) *auto*

lemma *is-Succ-normal-constrs-atom*:

$\forall a \in set (constrs-atom n a). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
by (*cases a*) (*use is-Succ-normal-constrs-term in auto*)

lemma *is-Succ-normal-constrs-fm*:

$\forall a \in set (constrs-fm n \varphi). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
by (*induction \varphi*) (*use is-Succ-normal-constrs-atom in auto*)

lemma *is-Var-Eq-Zero-if-is-NEq-constrs-term*:

$\forall a \in set (constrs-term n t). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq (SVar x) Zero)$
by (*induction t*) *auto*

lemma *is-Var-Eq-Zero-if-is-NEq-constrs-atom*:

$\forall a \in set (constrs-atom n a). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq (SVar x) Zero)$
by (*cases a*) (*use is-Var-Eq-Zero-if-is-NEq-constrs-term in auto*)

lemma *is-Var-Eq-Zero-if-is-NEq-constrs-fm*:

$\forall a \in set (constrs-fm n \varphi). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq (SVar x) Zero)$
by (*induction \varphi*) (*use is-Var-Eq-Zero-if-is-NEq-constrs-atom in auto*)

lemma *types-term-if-I-atom-constrs-term*:

includes *no member-ASCII-syntax*

assumes $(\forall e \in \text{set } (\text{constrs-term } n \ t)). \text{MLSS-Suc-Theory.I-atom } v \ e)$
shows $(\lambda x. v \ (n \ (\text{Var } x))) \vdash t : v \ (n \ t)$
using *assms*
by $(\text{induction } t) \ (\text{auto intro: types-pset-term.intros})$

lemma *types-pset-atom-if-I-atom-constrs-atom:*

fixes $a :: 'a \ \text{pset-atom}$
assumes $(\forall e \in \text{set } (\text{constrs-atom } n \ a)). \text{MLSS-Suc-Theory.I-atom } v \ e)$
shows $(\lambda x. v \ (n \ (\text{Var } x))) \vdash a$
using *assms*
by $(\text{cases } a)$
(auto simp: types-pset-atom.simps ball-Un dest!: types-term-if-I-atom-constrs-term)

lemma *types-pset-fm-if-I-atom-constrs-fm:*

fixes $\varphi :: 'a \ \text{pset-fm}$
assumes $(\forall e \in \text{set } (\text{constrs-fm } n \ \varphi)). \text{MLSS-Suc-Theory.I-atom } v \ e)$
shows $(\lambda x. v \ (n \ (\text{Var } x))) \vdash \varphi$
using *assms*
by $(\text{induction } \varphi)$
(auto intro: types-fmI types-pset-atom-if-I-atom-constrs-atom)

lemma *I-atom-constrs-term-if-types-term:*

includes *no member-ASCII-syntax*
assumes *inj-on* $n \ T$ *subterms* $t \subseteq T$
assumes $v \vdash t : k$
shows $(\forall e \in \text{set } (\text{constrs-term } n \ t)).$
MLSS-Suc-Theory.I-atom $(\lambda x. \text{type-of-term } v \ (\text{inv-into } T \ n \ x)) \ e)$
using *assms inv-into-f-f[OF assms(1) subsetD[OF assms(2)]]*
by $(\text{induction } t \ \text{arbitrary: } T \ k)$
(auto elim!: types-pset-term-cases intro!: type-of-term-if-types-term simp: type-of-term-if-types-term)

lemma *I-atom-constrs-atom-if-types-pset-atom:*

fixes $a :: 'a \ \text{pset-atom}$
assumes *inj-on* $n \ T$ *subterms* $a \subseteq T$
assumes $v \vdash a$
shows $(\forall e \in \text{set } (\text{constrs-atom } n \ a)).$
MLSS-Suc-Theory.I-atom $(\lambda x. \text{type-of-term } v \ (\text{inv-into } T \ n \ x)) \ e)$
using *assms I-atom-constrs-term-if-types-term*
by $(\text{cases } a)$
(force simp: types-pset-atom.simps type-of-term-if-types-term subsetD)+

lemma *I-atom-constrs-fm-if-types-pset-fm:*

fixes $\varphi :: 'a \ \text{pset-fm}$
assumes *inj-on* $n \ T$ *subterms* $\varphi \subseteq T$
assumes $v \vdash \varphi$
shows $(\forall e \in \text{set } (\text{constrs-fm } n \ \varphi)).$
MLSS-Suc-Theory.I-atom $(\lambda x. \text{type-of-term } v \ (\text{inv-into } T \ n \ x)) \ e)$
using *assms*

by (*induction* φ)
(*auto dest: types-fmD simp: I-atom-constrs-atom-if-types-pset-atom*)

lemma *inv-into-f-eq-if-subs*:
assumes *inj-on* $f B A \subseteq B y \in f \text{ ' } A$
shows *inv-into* $B f y = \text{inv-into } A f y$
using *assms inv-into-f-eq*
by (*metis f-inv-into-f inv-into-into subset-eq*)

lemma *UN-set-suc-atom-constrs-term-eq-image-subterms*:
 $\bigcup (\text{set-suc-atom ' set (constrs-term } n t)) = n \text{ ' subterms } t$
by (*induction t auto*)

lemma *UN-set-suc-atom-constrs-atom-eq-image-subterms*:
 $\bigcup (\text{set-suc-atom ' set (constrs-atom } n a)) = n \text{ ' subterms } a$
by (*induction a auto simp: UN-set-suc-atom-constrs-term-eq-image-subterms*)

lemma *UN-set-suc-atom-constrs-fm-eq-image-subterms*:
 $\bigcup (\text{set-suc-atom ' set (constrs-fm } n \varphi)) = n \text{ ' subterms } \varphi$
by (*induction \varphi auto simp: UN-set-suc-atom-constrs-atom-eq-image-subterms*)

lemma
fixes $\varphi :: \text{'a pset-fm}$
assumes *inj-on* $n (\text{subterms } \varphi)$
assumes *ssolve* (*MLSS-Suc-Theory.elim-NEq-Zero* (*constrs-fm* $n \varphi$)) = *Some ss*
shows *types-pset-fm-assign-solve*: $(\lambda x. \text{sassign } ss (n (\text{Var } x))) \vdash \varphi$
and *minimal-assign-solve*: $\llbracket v \vdash \varphi; z \in \text{vars } \varphi \rrbracket \implies \text{sassign } ss (n (\text{Var } z)) \leq$

$v z$

proof –

note *I-atom-assign-if-solve-elim-NEq-Zero-Some*[*OF - - assms(2)*]
then have $\forall e \in \text{set (constrs-fm } n \varphi). \text{MLSS-Suc-Theory.I-atom (sassign } ss) e$
using *is-Succ-normal-constrs-fm is-Var-Eq-Zero-if-is-NEq-constrs-fm* **by** *blast*
note *types-pset-fm-if-I-atom-constrs-fm*[*OF this*]
then show $(\lambda x. \text{sassign } ss (n (\text{Var } x))) \vdash \varphi$.

let $?v' = \lambda x. \text{type-of-term } v (\text{inv-into (subterms } \varphi) n x)$

note *I-atom-assign-minimal-if-solve-elim-NEq-Zero-Some*[*OF - - assms(2)*]

then have *assign-leq*: $\text{sassign } ss z \leq v z$

if $\forall a \in \text{set (constrs-fm } n \varphi). \text{MLSS-Suc-Theory.I-atom } v a$

$z \in \bigcup (\text{set-suc-atom ' set (constrs-fm } n \varphi))$ **for** $v z$

using *that is-Succ-normal-constrs-fm is-Var-Eq-Zero-if-is-NEq-constrs-fm*

by *blast*

show $\text{sassign } ss (n (\text{Var } z)) \leq v z$ **if** $v \vdash \varphi z \in \text{vars } \varphi$

proof –

note *assign-leq*[*unfolded UN-set-suc-atom-constrs-fm-eq-image-subterms, where ?v=?v'*]

note *assign-leq'* = *this*[*OF I-atom-constrs-fm-if-types-pset-fm*[*OF assms(1) - <v \vdash \varphi, simplified*]]

```

from ⟨z ∈ vars φ⟩ have n (Var z) ∈ n ‘ subterms φ
by (simp add: vars-fm-subsubterms-fm)
from assign-leq'[OF this] ⟨inj-on n (subterms φ)⟩ ⟨z ∈ vars φ⟩ show ?thesis
using vars-fm-subsubterms-fm
by (metis inv-into-f-f type-of-term-if-types-term types-pset-term.intros(2))
qed
qed

```

```

lemma types-term-minimal:
includes no member-ASCII-syntax
assumes  $\bigwedge z. z \in \text{vars } t \implies v\text{-min } z \leq v z$ 
assumes  $v\text{-min} \vdash t : k' \ v \vdash t : k$ 
shows  $k' \leq k$ 
using assms
by (induction t arbitrary: k' k) (auto elim!: types-pset-term-cases)

```

```

lemma constra-term-subsubconstra-term:
assumes s ∈ subterms t
shows set (constra-term n s) ⊆ set (constra-term n t)
using assms
by (induction t) auto

```

```

lemma constra-term-subsubconstra-atom:
assumes t ∈ subterms a
shows set (constra-term n t) ⊆ set (constra-atom n a)
using assms constra-term-subsubconstra-term by (cases a) force+

```

```

lemma constra-term-subsubconstra-fm:
assumes t ∈ subterms φ
shows set (constra-term n t) ⊆ set (constra-fm n φ)
using assms
by (induction φ) (auto simp: constra-term-subsubconstra-atom)

```

```

lemma urelem-iff-assign-eq-0:
includes no member-ASCII-syntax
assumes inj-on n (subterms φ)
assumes t ∈ subterms φ
assumes  $ssolve (MLSS\text{-Suc}\text{-Theory.}\text{elim-NEq-Zero } (constra\text{-fm } n \ \varphi)) = \text{Some } ss$ 
shows  $urelem \ \varphi \ t \longleftrightarrow sassign \ ss \ (n \ t) = 0$ 

```

```

proof –
note types = types-pset-fm-assign-solve[OF assms(1,3)]

```

```

note I-atom-assign-if-solve-elim-NEq-Zero-Some[OF - - assms(3)]
then have  $\forall e \in \text{set } (constra\text{-fm } n \ \varphi). MLSS\text{-Suc}\text{-Theory.}\text{I-atom } (sassign \ ss) \ e$ 
using is-Succ-normal-constra-fm is-Var-Eq-Zero-if-is-NEq-constra-fm by blast
then have  $\forall e \in \text{set } (constra\text{-term } n \ t). MLSS\text{-Suc}\text{-Theory.}\text{I-atom } (sassign \ ss) \ e$ 
using constra-term-subsubconstra-fm[OF ⟨t ∈ subterms φ⟩] by blast
note type-term-t = types-term-if-I-atom-constra-term[OF this]

```

```

note minimal = minimal-assign-solve[OF assms(1,3)]
have  $\exists lt'. v \vdash t : lt' \wedge sassign\ ss\ (n\ t) \leq lt'$ 
  if  $v \vdash \varphi$  for  $v$ 
proof –
  from that obtain  $lt'$  where  $v \vdash t : lt'$ 
    using  $\langle t \in subterms\ \varphi \rangle$ 
    by (meson not-Some-eq subterms-type-pset-fm-not-None)
  moreover note minimal[OF that] types-term-minimal[OF - type-term-t]
  ultimately show ?thesis
    by (metis assms(2) mem-vars-fm-if-mem-subterms-fm)
qed

then show  $urelem\ \varphi\ t \longleftrightarrow sassign\ ss\ (n\ t) = 0$ 
  using types type-term-t types-term-unique unfolding urelem-def
  by (metis le-zero-eq)
qed

lemma not-types-fm-if-solve-eq-None:
  fixes  $\varphi :: 'a\ pset-fm$ 
  assumes inj-on  $n$  (subterms  $\varphi$ )
  assumes ssolve (MLSS-Suc-Theory.elim-NEq-Zero (constrs-fm  $n\ \varphi$ )) = None
  shows  $\neg v \vdash \varphi$ 
proof
  assume  $v \vdash \varphi$ 
  note I-atom-constrs-fm-if-types-pset-fm[OF assms(1) - this]
  moreover
  note not-I-atom-if-solve-elim-NEq-Zero-None[OF - - assms(2)]
  then have  $\exists a \in set\ (constrs-fm\ n\ \varphi). \neg MLSS-Suc-Theory.I-atom\ v\ a$  for  $v$ 
    using is-Succ-normal-constrs-fm is-Var-Eq-Zero-if-is-NEq-constrs-fm by blast
  ultimately show False
    by blast
qed

```

Chapter 4

Deciding MLSS

4.1 The Realisation Function

We define an abstract formulation of a model for membership relations. This is later used to define a model for open branches of an MLSS tableau.

abbreviation *parents* :: ('a,'b) pre-digraph \Rightarrow 'a \Rightarrow 'a set
where *parents* G $s \equiv \{u. u \rightarrow_G s\}$

abbreviation *ancestors* :: ('a,'b) pre-digraph \Rightarrow 'a \Rightarrow 'a set
where *ancestors* G $s \equiv \{u. u \rightarrow^+_G s\}$

lemma (in *fin-digraph*) *parents-sub-verts*: *parents* G $s \subseteq$ *verts* G
using *reachable-in-verts* by *blast*

lemma (in *fin-digraph*) *finite-parents[intro]*: *finite* (*parents* G s)
using *finite-subset[OF parents-sub-verts finite-verts]* .

lemma (in *fin-digraph*) *finite-ancestors[intro]*: *finite* (*ancestors* G s)
using *reachable-in-verts*
by (auto intro: rev-finite-subset[where ?A=ancestors G s , OF finite-verts])

lemma (in *wf-digraph*) *in-ancestors-if-dominates[simp, intro]*:
assumes $s \rightarrow_G t$
shows $s \in$ *ancestors* G t
using *assms* by *blast*

lemma (in *wf-digraph*) *ancestors-mono*:
assumes $s \in$ *ancestors* G t
shows *ancestors* G $s \subseteq$ *ancestors* G t
using *assms* by *fastforce*

locale *dag* = *digraph* G **for** G +
assumes *acyclic*: $\nexists c.$ *cycle* c
begin

```

lemma ancestors-asym:
  assumes  $s \in \text{ancestors } G \ t$ 
  shows  $t \notin \text{ancestors } G \ s$ 
proof
  assume  $t \in \text{ancestors } G \ s$ 
  then obtain  $p1 \ p2$  where  $\text{awalk } t \ p1 \ s \ p1 \neq [] \ \text{awalk } s \ p2 \ t \ p2 \neq []$ 
    using assms reachable1-awalk by auto
  then have  $\text{closed-w } (p1 \ @ \ p2)$ 
    unfolding closed-w-def by auto
  with closed-w-imp-cycle acyclic show False
    by blast
qed

```

```

lemma ancestors-strict-mono:
  assumes  $s \in \text{ancestors } G \ t$ 
  shows  $\text{ancestors } G \ s \subset \text{ancestors } G \ t$ 
  using assms ancestors-mono ancestors-asym by blast

```

```

lemma card-ancestors-strict-mono:
  assumes  $s \rightarrow_G \ t$ 
  shows  $\text{card } (\text{ancestors } G \ s) < \text{card } (\text{ancestors } G \ t)$ 
  using assms finite-ancestors ancestors-strict-mono
  by (metis in-ancestors-if-dominates psubset-card-mono)

```

end

The realisation assumes that the terms can be split into a set of base terms B that are realised with the function I and set terms T that are realised according to the structure of the membership relation (represented as a graph G).

```

locale realisation = dag G for G +
  fixes  $B \ T :: 'a \ \text{set}$ 
  fixes  $I :: 'a \Rightarrow \text{hf}$ 
  fixes  $\text{eq} :: 'a \ \text{rel}$ 
  assumes B-T-partition-verts:  $B \cap \ T = \{\} \ \text{verts } G = B \cup \ T$ 
  assumes P-urelems:  $\bigwedge p \ t. \ p \in \ B \Longrightarrow \neg \ t \rightarrow_G \ p$ 
begin

```

```

lemma
  shows finite-B:  $\text{finite } B$ 
  and finite-T:  $\text{finite } T$ 
  and finite-B-un-T:  $\text{finite } (B \cup \ T)$ 
  using finite-verts B-T-partition-verts by auto

```

```

abbreviation eq-class  $x \equiv \text{eq} \ \{x\}$ 

```

```

function realise  $:: 'a \Rightarrow \text{hf}$  where
   $x \in \ B \Longrightarrow \text{realise } x = \text{HF } (\text{realise } \ \text{'parents } G \ x) \sqcup \text{HF } (I \ \text{'eq-class } x)$ 
   $| \ t \in \ T \Longrightarrow \text{realise } t = \text{HF } (\text{realise } \ \text{'parents } G \ t)$ 

```

| $x \notin B \cup T \implies \text{realise } x = 0$
using *B-T-partition-verts* **by** *auto*
termination
by (*relation measure* ($\lambda t. \text{card} (\text{ancestors } G t)$)) (*simp-all add: card-ancestors-strict-mono*)

lemma *finite-realisation-parents*[*simp, intro!*]: *finite* (*realise* ' *parents* *G t*)
by *auto*

function *height* :: 'a \Rightarrow nat **where**
 $\forall s. \neg s \rightarrow_G t \implies \text{height } t = 0$
| $s \rightarrow_G t \implies \text{height } t = \text{Max} (\text{height} \text{ ' parents } G t) + 1$
using *P-urelems* **by** *force+*
termination
by (*relation measure* ($\lambda t. \text{card} (\text{ancestors } G t)$)) (*simp-all add: card-ancestors-strict-mono*)

lemma *height-cases'*:
obtains
(*Zero*) *height* *t = 0*
| (*Suc-Max*) *s* **where** $s \rightarrow_G t$ *height* *t = height* *s + 1*
apply(*cases* *t* *rule: height.cases*)
using *Max-in*[*OF finite-imageI*[**where** *?h=height*, *OF finite-parents*]]
by *auto*

lemma *lemma1-1*:
assumes $s \rightarrow_G t$
shows *height* *s < height* *t*
proof(*cases* *t* *rule: height-cases'*)
case *Zero*
with *assms* **show** *?thesis* **by** *simp*
next
case (*Suc-Max* *s'*)
note *finite-imageI*[**where** *?h=height*, *OF finite-parents*]
note *Max-ge*[*OF this*, *of height* *s t*]
with *assms* *Suc-Max* **show** *?thesis*
by *simp*
qed

lemma *dominates-if-mem-realisation*:
assumes $\bigwedge x y. I x \neq \text{realise } y$
assumes *realise* *s* \in *realise* *t*
obtains *s'* **where** $s' \rightarrow_G t$ *realise* *s = realise* *s'*
using *assms*(2-)
proof(*induction* *t* *rule: realise.induct*)
case (1 *x*)
with *assms*(1) **show** *?case*
by *simp* (*metis* (*no-types*, *lifting*) *image-iff mem-Collect-eq*)
qed *auto*

lemma (**in** -) *Max-le-if-All-Ex-le*:

```

assumes finite A finite B
  and  $A \neq \{\}$ 
  and  $\forall a \in A. \exists b \in B. a \leq b$ 
  shows  $\text{Max } A \leq \text{Max } B$ 
using assms
proof(induction rule: finite-induct)
  case (insert a A)
  then obtain  $b B'$  where  $B = \text{insert } b B'$ 
    by (metis equals0I insert-absorb)
  with insert show ?case
    by (meson Max-ge-iff Max-in finite.insertI insert-not-empty)
qed blast

lemma lemma1-2':
  assumes  $\bigwedge x y. I x \neq \text{realise } y$ 
  assumes  $t1 \in B \cup T \ t2 \in B \cup T \ \text{realise } t1 = \text{realise } t2$ 
  shows  $\text{height } t1 \leq \text{height } t2$ 
  using assms(2-)
proof(induction height t1 arbitrary: t1 t2 rule: less-induct)
  case less
  then show ?case
  proof(cases height t1)
    case (Suc h)
    then obtain  $s1$  where  $s1 \rightarrow_G t1$ 
      by (cases t1 rule: height.cases) auto
    have Ex-approx:  $\exists v. v \rightarrow_G t2 \wedge \text{realise } w = \text{realise } v \ \text{if } w \rightarrow_G t1 \ \text{for } w$ 
    proof –
      from that less(2) have  $\text{realise } w \in \text{realise } t1$ 
        by (induction t1 rule: realise.induct) auto
      with less.prem1 have  $\text{realise } w \in \text{realise } t2$ 
        by metis
      with dominates-if-mem-realisation assms(1) obtain  $v$ 
        where  $v \rightarrow_G t2 \ \text{realise } w = \text{realise } v$ 
        by blast
      with less.prem1 show ?thesis
        by auto
    qed
  moreover
  have  $\text{height } w \leq \text{height } v$ 
  if  $w \rightarrow_G t1 \ v \in B \cup T \ \text{realise } w = \text{realise } v \ \text{for } w \ v$ 
  proof –
    have  $w \in B \cup T$ 
      using adj-in-verts[OF that(1)] B-T-partition-verts(2) by blast
    from less.hyps[OF lemma1-1, OF that(1) this that(2,3)] show ?thesis .
  qed
  ultimately have IH':  $\exists v \in \text{parents } G \ t2. \ \text{height } w \leq \text{height } v$ 
  if  $w \in \text{parents } G \ t1 \ \text{for } w$ 
    by (metis that adj-in-verts(1) mem-Collect-eq B-T-partition-verts(2))
  then have Max-le:  $\text{Max } (\text{height } \text{'parents } G \ t1) \leq \text{Max } (\text{height } \text{'parents } G \ t2)$ 

```

```

proof –
  have finite (height ‘ parents G t1)
    finite (height ‘ parents G t2)
    height ‘ parents G t1 ≠ {}
  using finite-parents ⟨s1 →G t1⟩ by blast+
  with Max-le-if-All-Ex-le[OF this] IH' show ?thesis by blast
qed
show ?thesis
proof –
  note s1 = ⟨s1 →G t1⟩
  with Ex-approx obtain s2 where s2 →G t2
    by blast
  with s1 Max-le show ?thesis by simp
qed
qed simp
qed

```

```

lemma lemma1-2:
  assumes  $\bigwedge x y. I x \neq \text{realise } y$ 
  assumes  $t1 \in B \cup T \ t2 \in B \cup T \ \text{realise } t1 = \text{realise } t2$ 
  shows  $\text{height } t1 = \text{height } t2$ 
  using assms lemma1-2' le-antisym by metis

```

```

lemma lemma1-3:
  assumes  $\bigwedge x y. I x \neq \text{realise } y$ 
  assumes  $s \in B \cup T \ t \in B \cup T \ \text{realise } s \in \text{realise } t$ 
  shows  $\text{height } s < \text{height } t$ 
proof –
  from dominates-if-mem-realisation[OF assms(1,4)] obtain s'
    where  $s' \rightarrow_G t \ \text{realise } s' = \text{realise } s$ 
    by metis
  then have  $\text{height } s = \text{height } s'$ 
    using lemma1-2[OF assms(1,2)]
    by (metis adj-in-verts(1) B-T-partition-verts(2))
  also have  $\dots < \text{height } t$ 
    using ⟨s' →G t⟩ lemma1-1 by blast
  finally show ?thesis .
qed

```

end

```

theory MLSS-HF-Extras
  imports HereditarilyFinite.Rank
begin

```

```

lemma hcard-ord-of[simp]:
  hcard (ord-of n) = n
  unfolding hcard-def hfset-ord-of card-image[OF inj-ord-of]
  by simp

```

lemma *hcard-HF*: *finite A* \implies *hcard (HF A) = card A*
unfolding *hcard-def* **using** *hfset-HF* **by** *simp*

end

4.2 A Decision Procedure for MLSS

This theory proves the soundness and completeness of the tableau calculus defined above. It then lifts those properties to a recursive procedure that applies the rules of the calculus exhaustively. To obtain a decision procedure, we also prove termination.

4.2.1 Basic Definitions

definition *lin-sat* $b \equiv \forall b'. \text{lexpands } b' b \longrightarrow \text{set } b' \subseteq \text{set } b$

lemma *lin-satD*:
assumes *lin-sat b*
assumes *lexpands b' b*
assumes $x \in \text{set } b'$
shows $x \in \text{set } b$
using *assms* **unfolding** *lin-sat-def* **by** *auto*

lemma *not-lin-satD*: $\neg \text{lin-sat } b \implies \exists b'. \text{lexpands } b' b \wedge \text{set } b \subset \text{set } (b' @ b)$
unfolding *lin-sat-def* **by** *auto*

definition *sat* $b \equiv \text{lin-sat } b \wedge (\nexists bs'. \text{bexpands } bs' b)$

lemma *satD*:
assumes *sat b*
shows $\text{lin-sat } b \wedge \nexists bs'. \text{bexpands } bs' b$
using *assms* **unfolding** *sat-def* **by** *auto*

definition *wits* :: '*a* branch \Rightarrow '*a* set **where**
wits b $\equiv \text{vars } b - \text{vars } (\text{last } b)$

definition *pwits* :: '*a* branch \Rightarrow '*a* set **where**
pwits b $\equiv \{c \in \text{wits } b. \forall t \in \text{subterms } (\text{last } b). \\ AT (Var c =_s t) \notin \text{set } b \wedge AT (t =_s Var c) \notin \text{set } b\}$

lemma *wits-singleton[simp]*: *wits* $[\varphi] = \{\}$
unfolding *wits-def* *vars-branch-simps* **by** *simp*

lemma *pwits-singleton[simp]*: *pwits* $[\varphi] = \{\}$
unfolding *pwits-def* **by** *auto*

lemma *pwitsD*:

```

assumes  $c \in pwits\ b$ 
shows  $c \in wits\ b$ 
   $t \in subterms\ (last\ b) \implies AT\ (Var\ c =_s\ t) \notin set\ b$ 
   $t \in subterms\ (last\ b) \implies AT\ (t =_s\ Var\ c) \notin set\ b$ 
using assms unfolding pwits-def by auto

lemma pwitsI:
assumes  $c \in wits\ b$ 
assumes  $\bigwedge t. t \in subterms\ (last\ b) \implies AT\ (Var\ c =_s\ t) \notin set\ b$ 
assumes  $\bigwedge t. t \in subterms\ (last\ b) \implies AT\ (t =_s\ Var\ c) \notin set\ b$ 
shows  $c \in pwits\ b$ 
using assms unfolding pwits-def by blast

lemma finite-wits: finite (wits  $b$ )
unfolding wits-def using finite-vars-branch by auto

lemma finite-pwits: finite (pwits  $b$ )
proof –
  have  $pwits\ b \subseteq wits\ b$ 
    unfolding pwits-def by simp
  then show ?thesis using finite-wits finite-subset by blast
qed

lemma lexpands-subterms-branch-eq:
   $lexpands\ b'\ b \implies b \neq [] \implies subterms\ (b' @ b) = subterms\ b$ 
proof(induction rule: lexpands.induct)
  case ( $1\ b'\ b$ )
    then show ?case
      apply(induction rule: lexpands-fm.induct)
      apply(auto simp: subterms-branch-def)
    done
  next
  case ( $2\ b'\ b$ )
    then show ?case
      apply(induction rule: lexpands-un.induct)
      using subterms-branch-subterms-subterms-fm-trans[OF - subterms-refl]
      apply(auto simp: subterms-branch-simps)
      intro: subterms-term-subterms-branch-trans
      dest: subterms-branchD AT-mem-subterms-branchD AF-mem-subterms-branchD)
    done
  next
  case ( $3\ b'\ b$ )
    then show ?case
      apply(induction rule: lexpands-int.induct)
      using subterms-branch-subterms-subterms-fm-trans[OF - subterms-refl]
      apply(auto simp: subterms-branch-simps)
      intro: subterms-term-subterms-branch-trans
      dest: subterms-branchD AT-mem-subterms-branchD AF-mem-subterms-branchD)
    done

```

```

next
  case (4 b' b)
  then show ?case
    apply(induction rule: lexpands-diff.induct)
    using subterms-branch-subterms-subterms-fm-trans[OF - subterms-refl]
      apply(auto simp: subterms-branch-simps
        intro: subterms-term-subterms-branch-trans
        dest: subterms-branchD AT-mem-subterms-branchD AF-mem-subterms-branchD)
    done
next
  case (5 b' b)
  then show ?case
  proof(induction rule: lexpands-single.induct)
    case (1 t1 b)
    then show ?case
      using subterms-branch-subterms-subterms-fm-trans[OF - subterms-refl]
      apply(auto simp: subterms-branch-simps
        dest: subterms-fmD intro: subterms-term-subterms-branch-trans)
    done
  qed (auto simp: subterms-branch-simps subterms-term-subterms-atom-Atom-trans
    dest: subterms-branchD AF-mem-subterms-branchD
    intro: subterms-term-subterms-branch-trans)
next
  case (6 b' b)
  have *: subterms-atom (subst-tlvl t1 t2 a)  $\subseteq$  subterms t2  $\cup$  subterms-atom a
    for t1 t2 and a :: 'a pset-atom
  by (cases (t1, t2, a) rule: subst-tlvl.cases) auto
  from 6 show ?case
  by (induction rule: lexpands-eq.induct)
    (auto simp: subterms-branch-def subterms-term-subterms-atom-Atom-trans
    dest!: subsetD[OF *])
qed

lemma lexpands-vars-branch-eq:
  lexpands b' b  $\implies$  b  $\neq$  []  $\implies$  vars (b' @ b) = vars b
  using lexpands-subterms-branch-eq subterms-branch-eq-if-vars-branch-eq by metis

lemma bexpands-nowit-subterms-branch-eq:
  bexpands-nowit bs' b  $\implies$  b'  $\in$  bs'  $\implies$  b  $\neq$  []  $\implies$  subterms (b' @ b) = subterms b
  proof(induction rule: bexpands-nowit.induct)
    case (3 s t1 t2 b)
    then show ?case
    by (auto simp: subterms-term-subterms-atom-Atom-trans subterms-branch-simps)
  next
  case (4 s t1 b t2)
  then show ?case
    using subterms-branch-subterms-subterms-fm-trans[OF - - 4(2)]
    by (auto simp: subterms-term-subterms-atom-Atom-trans subterms-branch-simps)
  next

```

case (5 s t1 b t2)
then show ?case
using subterms-branch-subterms-subterms-fm-trans[OF - - 5(2)]
by (auto simp: subterms-term-subterms-atom-Atom-trans subterms-branch-simps)
qed (use subterms-branch-def in ⟨(fastforce simp: subterms-branch-simps)+⟩)

lemma bexpands-nowit-vars-branch-eq:
 $bexpands\ nowit\ bs'\ b \implies b' \in bs' \implies b \neq [] \implies vars\ (b' @ b) = vars\ b$
using bexpands-nowit-subterms-branch-eq subterms-branch-eq-if-vars-branch-eq
by metis

lemma lexpands-wits-eq:
 $lexpands\ b'\ b \implies b \neq [] \implies wits\ (b' @ b) = wits\ b$
using lexpands-vars-branch-eq unfolding wits-def **by** force

lemma bexpands-nowit-wits-eq:
assumes bexpands-nowit bs' b b' ∈ bs' b ≠ []
shows wits (b' @ b) = wits b
using bexpands-nowit-vars-branch-eq[OF assms] assms(3)
unfolding wits-def **by** simp

lemma bexpands-wit-vars-branch-eq:
assumes bexpands-wit t1 t2 x bs' b b' ∈ bs' b ≠ []
shows vars (b' @ b) = insert x (vars b)
using assms bexpands-witD[OF assms(1)]
by (auto simp: mem-vars-fm-if-mem-subterms-fm vars-branch-simps vars-fm-vars-branchI)

lemma bexpands-wit-wits-eq:
assumes bexpands-wit t1 t2 x bs' b b' ∈ bs' b ≠ []
shows wits (b' @ b) = insert x (wits b)
using assms bexpands-witD[OF assms(1)]
unfolding wits-def
by (auto simp: mem-vars-fm-if-mem-subterms-fm vars-branch-simps vars-branch-def)

lemma lexpands-pwits-subs:
assumes lexpands b' b b ≠ []
shows pwits (b' @ b) ⊆ pwits b
using assms lexpands-wits-eq[OF assms]
by (induction rule: lexpands-induct) (auto simp: pwits-def)

no-new-subterms

definition no-new-subterms b ≡
 $\forall t \in \text{subterms } b. t \notin \text{Var } 'wits\ b \implies t \in \text{subterms } (\text{last } b)$

lemma no-new-subtermsI:
assumes $\bigwedge t. t \in \text{subterms } b \implies t \notin \text{Var } 'wits\ b \implies t \in \text{subterms } (\text{last } b)$
shows no-new-subterms b

using *assms* **unfolding** *no-new-subterms-def* **by** *blast*

lemma *Var-mem-subterms-branch-and-not-in-wits*:

assumes $Var\ v \in subterms\ b\ v \notin wits\ b$

shows $v \in vars\ (last\ b)$

using *assms* **unfolding** *wits-def no-new-subterms-def*

by (*auto simp: image-set-diff[unfolded inj-on-def] image-UN*

Un-vars-term-subterms-branch-eq-vars-branch[symmetric])

lemma *subterms-branch-cases*:

assumes $t \in subterms\ b\ t \notin Var\ 'wits\ b$

obtains

(*Empty*) n **where** $t = \emptyset\ n$

| (*Union*) $t1\ t2$ **where** $t = t1 \sqcup_s t2$

| (*Inter*) $t1\ t2$ **where** $t = t1 \sqcap_s t2$

| (*Diff*) $t1\ t2$ **where** $t = t1 -_s t2$

| (*Single*) $t1$ **where** $t = Single\ t1$

| (*Var*) v **where** $t = Var\ v\ v \in vars\ (last\ b)$

proof(*cases t*)

case (*Var x*)

with *assms* **have** $x \in vars\ (last\ b)$

using *Var-mem-subterms-branch-and-not-in-wits* **by** (*metis imageI*)

with *Var* **that** **show** *?thesis* **by** *blast*

qed (*use assms that in auto*)

lemma *no-new-subterms-casesI[case-names Empty Union Inter Diff Single]*:

assumes $\bigwedge n. \emptyset\ n \in subterms\ b \implies \emptyset\ n \in subterms\ (last\ b)$

assumes $\bigwedge t1\ t2. t1 \sqcup_s t2 \in subterms\ b \implies t1 \sqcup_s t2 \in subterms\ (last\ b)$

assumes $\bigwedge t1\ t2. t1 \sqcap_s t2 \in subterms\ b \implies t1 \sqcap_s t2 \in subterms\ (last\ b)$

assumes $\bigwedge t1\ t2. t1 -_s t2 \in subterms\ b \implies t1 -_s t2 \in subterms\ (last\ b)$

assumes $\bigwedge t. Single\ t \in subterms\ b \implies Single\ t \in subterms\ (last\ b)$

shows *no-new-subterms b*

proof(*intro no-new-subtermsI*)

fix t **assume** $t \in subterms\ b\ t \notin Var\ 'wits\ b$

with *this assms* **show** $t \in subterms\ (last\ b)$

by (*cases t rule: subterms-branch-cases*) (*auto simp: vars-fm-sub-subterms-fm*)

qed

lemma *no-new-subtermsD*:

assumes *no-new-subterms b*

shows $\bigwedge n. \emptyset\ n \in subterms\ b \implies \emptyset\ n \in subterms\ (last\ b)$

$\bigwedge t1\ t2. t1 \sqcup_s t2 \in subterms\ b \implies t1 \sqcup_s t2 \in subterms\ (last\ b)$

$\bigwedge t1\ t2. t1 \sqcap_s t2 \in subterms\ b \implies t1 \sqcap_s t2 \in subterms\ (last\ b)$

$\bigwedge t1\ t2. t1 -_s t2 \in subterms\ b \implies t1 -_s t2 \in subterms\ (last\ b)$

$\bigwedge t. Single\ t \in subterms\ b \implies Single\ t \in subterms\ (last\ b)$

using *assms* **unfolding** *no-new-subterms-def* **by** *auto*

lemma *lexpands-no-new-subterms*:

assumes *lexpands b' b b \neq []*

```

assumes no-new-subterms b
shows no-new-subterms (b' @ b)
using assms unfolding no-new-subterms-def
by (simp add: lexpands-wits-eq lexpands-subterms-branch-eq[OF assms(1,2)])

lemma subterms-branch-subterms-atomI:
assumes Atom l ∈ set b t ∈ subterms-atom l
shows t ∈ subterms-branch b
using assms unfolding subterms-branch-def
apply (cases l rule: subterms-atom.cases)
apply (metis subterms-branch-def subterms-term-subterms-atom-Atom-trans sub-
terms-refl)+
done

lemma bexpands-nowit-no-new-subterms:
assumes bexpands-nowit bs' b b' ∈ bs' b ≠ []
assumes no-new-subterms b
shows no-new-subterms (b' @ b)
using assms unfolding no-new-subterms-def
by (simp add: bexpands-nowit-wits-eq bexpands-nowit-subterms-branch-eq[OF assms(1,2)])

lemma bexpands-wit-no-new-subterms:
assumes bexpands-wit t1 t2 x bs' b b ≠ [] b' ∈ bs'
assumes no-new-subterms b
shows no-new-subterms (b' @ b)
using assms
apply(induction rule: bexpands-wit.induct)
apply(auto simp: subterms-branch-simps
subterms-term-subterms-atom-trans subterms-term-subterms-fm-trans
elim: no-new-subtermsD
intro!: no-new-subterms-casesI)
done

lemma bexpands-no-new-subterms:
assumes bexpands bs' b b ≠ [] b' ∈ bs'
assumes no-new-subterms b
shows no-new-subterms (b' @ b)
using assms
apply(cases rule: bexpands.cases)
using bexpands-nowit-no-new-subterms bexpands-wit-no-new-subterms by metis+

lemma expandss-no-new-subterms:
assumes expandss b' b b ≠ [] no-new-subterms b
shows no-new-subterms b'
using assms
apply(induction b' b rule: expandss.induct)
using expandss-suffix suffix-bot.extremum-uniqueI
using lexpands-no-new-subterms bexpands-no-new-subterms
by blast+

```

lemmas *subterms-branch-subterms-fm-lastI* =
subterms-branch-subterms-subterms-fm-trans[*OF* - *subterms-refl*]

wits-subterms

definition *wits-subterms* :: 'a branch \Rightarrow 'a pset-term set **where**
wits-subterms b \equiv Var ' wits b \cup subterms (last b)

lemma *subterms-branch-eq-if-no-new-subterms*:

assumes *no-new-subterms* b b \neq []

shows *subterms-branch* b = *wits-subterms* b

using *assms no-new-subtermsD*[*OF* *assms*(1)]

proof –

note *simps* = *wits-def no-new-subterms-def wits-subterms-def*

subterms-branch-simps vars-branch-simps

with *assms* **show** ?thesis

by (*auto simp: simps vars-fm-subs-subterms-fm*

vars-branch-subs-subterms-branch[*unfolded image-subset-iff*]

intro: subterms-branch-subterms-fm-lastI)

qed

lemma *wits-subterms-last-disjnt*: Var ' wits b \cap subterms (last b) = {}

by (*auto simp: wits-def intro!: mem-vars-fm-if-mem-subterms-fm*)

4.2.2 Completeness of the Calculus

Proof of Lemma 2

fun *is-literal* :: 'a fm \Rightarrow bool **where**

is-literal (Atom -) = True

| *is-literal* (Neg (Atom -)) = True

| *is-literal* - = False

lemma *lexpands-no-wits-if-not-literal*:

defines P \equiv ($\lambda b. \forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$)

assumes *lexpands* b' b b \neq []

assumes P b

shows P (b' @ b)

using *assms*(2–) *lexpands-wits-eq*[*OF* *assms*(2,3)]

by (*induction rule: lexpands-induct*) (*auto simp: disjoint-iff P-def*)

lemma *bexpands-nowit-no-wits-if-not-literal*:

defines P \equiv ($\lambda b. \forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$)

assumes *bexpands-nowit* bs' b b' \in bs' b \neq []

assumes P b

shows P (b' @ b)

using *assms*(2–)

by (*induction rule: bexpands-nowit.induct*)

(*auto simp: Int-def P-def wits-def vars-fm-vars-branchI*)

lemma *bexpands-wit-no-wits-if-not-literal*:
defines $P \equiv (\lambda b. \forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\})$
assumes *bexpands-wit* $t1\ t2\ x\ bs'\ b\ b' \in bs'\ b \neq []$
assumes $P\ b$
shows $P\ (b' @ b)$
using *assms*(2-)
by (*induction rule: bexpands-wit.induct*)
(auto simp: Int-def P-def wits-def vars-fm-vars-branchI)

lemma *bexpands-no-wits-if-not-literal*:
defines $P \equiv (\lambda b. \forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\})$
assumes *bexpands* $bs'\ b\ b' \in bs'\ b \neq []$
assumes $P\ b$
shows $P\ (b' @ b)$
using *assms*(2-)
apply(*cases* $bs'\ b$ *rule: bexpands-cases*)
using *bexpands-wit-no-wits-if-not-literal bexpands-nowit-no-wits-if-not-literal*
using P -*def* **by** *fast+*

lemma *expandss-no-wits-if-not-literal*:
defines $P \equiv (\lambda b. \forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\})$
assumes *expandss* $b'\ b\ b \neq []$
assumes $P\ b$
shows $P\ b'$
using *assms*(2-)
apply (*induction rule: expandss.induct*)
using *lexpands-no-wits-if-not-literal bexpands-no-wits-if-not-literal*
apply (*metis P-def expandss-suffix suffix-bot.extremum-uniqueI*)
done

lemma *lexpands-fm-pwits-eq*:
assumes *lexpands-fm* $b'\ b\ b \neq []$
assumes $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$
shows $\text{pwits } (b' @ b) = \text{pwits } b$
using *assms*
apply(*induction rule: lexpands-fm.induct*)
apply(*fastforce simp: pwits-def wits-def vars-branch-def*)
done

lemma *lexpands-un-pwits-eq*:
assumes *lexpands-un* $b'\ b\ b \neq []$
assumes $\forall c \in \text{pwits } b. \forall t \in \text{wits-subterms } b.$
 $AT\ (\text{Var } c =_s t) \notin \text{set } b \wedge AT\ (t =_s \text{Var } c) \notin \text{set } b \wedge AT\ (t \in_s \text{Var } c) \notin \text{set } b$
shows $\text{pwits } (b' @ b) = \text{pwits } b$
proof –
note *lexpands.intros*(2)[*OF* *assms*(1)]
note *lexpands-wits-eq*[*OF* *this* $\langle b \neq [] \rangle$]
from *assms* *this* **have** $x \in \text{pwits } (b' @ b)$ **if** $x \in \text{pwits } b$ **for** x

using *that*
by (*induction rule: lexpands-un.induct*)
(auto simp: wits-subterms-def pwitsD intro!: pwitsI)
with *lexpands-pwits Subs [OF ‹lexpands b' b› ‹b ≠ []›]* **show** *?thesis*
by *auto*
qed

lemma *lexpands-int-pwits-eq*:
assumes *lexpands-int b' b b ≠ []*
assumes $\forall c \in \text{pwits } b. \forall t \in \text{wits-subterms } b.$
 $AT (Var\ c =_s\ t) \notin \text{set } b \wedge AT (t =_s\ Var\ c) \notin \text{set } b \wedge AT (t \in_s\ Var\ c) \notin \text{set } b$
shows $\text{pwits } (b' @ b) = \text{pwits } b$
proof –
note *lexpands.intros(3) [OF assms(1)]*
note *lexpands-wits-eq [OF this ‹b ≠ []›]*
from *assms this* **have** $x \in \text{pwits } (b' @ b)$ **if** $x \in \text{pwits } b$ **for** x
using *that*
by (*induction rule: lexpands-int.induct*)
(auto simp: wits-subterms-def pwitsD intro!: pwitsI)
with *lexpands-pwits Subs [OF ‹lexpands b' b› ‹b ≠ []›]* **show** *?thesis*
by *auto*
qed

lemma *lexpands-diff-pwits-eq*:
assumes *lexpands-diff b' b b ≠ []*
assumes $\forall c \in \text{pwits } b. \forall t \in \text{wits-subterms } b.$
 $AT (Var\ c =_s\ t) \notin \text{set } b \wedge AT (t =_s\ Var\ c) \notin \text{set } b \wedge AT (t \in_s\ Var\ c) \notin \text{set } b$
shows $\text{pwits } (b' @ b) = \text{pwits } b$
proof –
note *lexpands.intros(4) [OF assms(1)]*
note *lexpands-wits-eq [OF this ‹b ≠ []›]*
from *assms this* **have** $x \in \text{pwits } (b' @ b)$ **if** $x \in \text{pwits } b$ **for** x
using *that*
by (*induction rule: lexpands-diff.induct*)
(auto simp: wits-subterms-def pwitsD intro!: pwitsI)
with *lexpands-pwits Subs [OF ‹lexpands b' b› ‹b ≠ []›]* **show** *?thesis*
by *auto*
qed

lemma *bexpands-nowit-pwits-eq*:
assumes *bexpands-nowit bs' b b' ∈ bs' b b' ≠ []*
assumes $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$
shows $\text{pwits } (b' @ b) = \text{pwits } b$
using *assms*
proof –
from *assms* **have** $x \in \text{pwits } (b' @ b)$ **if** $x \in \text{pwits } b$ **for** x
using *that bexpands-nowit-wits-eq [OF assms(1–3)]*
by (*induction rule: bexpands-nowit.induct*)
(intro pwitsI; fastforce simp: pwitsD)+

moreover from *assms* **have** $\text{pwits } (b' @ b) \subseteq \text{pwits } b$
unfolding *pwits-def*
using *beexpands-nowit-wits-eq* **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

lemma *beexpands-wit-pwits-eq*:
assumes *beexpands-wit* $t1\ t2\ x\ bs'\ b\ b' \in bs'\ b \neq []$
shows $\text{pwits } (b' @ b) = \text{insert } x (\text{pwits } b)$
using *assms(2,3)* *beexpands-witD[OF assms(1)]*
unfolding *pwits-def* *beexpands-wit-wits-eq[OF assms]*
by (*auto simp: vars-fm-vars-branchI*)

lemma *lemma-2-lexpands*:
defines $P \equiv (\lambda b\ c\ t.\ AT\ (Var\ c =_s\ t) \notin \text{set } b \wedge AT\ (t =_s\ Var\ c) \notin \text{set } b \wedge AT\ (t \in_s\ Var\ c) \notin \text{set } b)$
assumes *lexpands* $b'\ b\ b \neq []$
assumes *no-new-subterms* b
assumes $\forall \varphi \in \text{set } b.\ \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$
assumes $\forall c \in \text{pwits } b.\ \forall t \in \text{wits-subterms } b.\ P\ b\ c\ t$
shows $\forall c \in \text{pwits } (b' @ b).\ \forall t \in \text{wits-subterms } (b' @ b).\ P\ (b' @ b)\ c\ t$
using *assms(2-6)*
using *lexpands-wits-eq[OF assms(2,3)]*
lexpands-pwits-subs[OF assms(2,3)]
proof(*induction rule: lexpands.induct*)
case ($1\ b'\ b$)

have $P\ (b' @ b)\ c\ t$
if $\forall \varphi \in \text{set } b'.\ \text{vars } \varphi \cap \text{wits } (b' @ b) = \{\}$
and $c \in \text{pwits } b\ t \in \text{wits-subterms } (b' @ b)$ **for** $c\ t$
proof –
from *that 1.prem5*
have $\forall \varphi \in \text{set } b'.\ \varphi \neq AT\ (Var\ c =_s\ t) \wedge \varphi \neq AT\ (t =_s\ Var\ c) \wedge \varphi \neq AT\ (t \in_s\ Var\ c)$
by (*auto simp: pwits-def disjoint-iff*)
with 1 **that show** *?thesis*
unfolding *P-def* *wits-subterms-def*
by (*metis Un-iff last-appendR set-append*)
qed
moreover from $1(1,4,6)$ **have** $\forall \varphi \in \text{set } b'.\ \text{vars } \varphi \cap \text{wits } (b' @ b) = \{\}$
by (*induction rule: lexpands-fm.induct*) (*auto simp: disjoint-iff*)
ultimately show *?case*
using 1 *lexpands-fm-pwits-eq* **by** *blast*

next
case ($2\ b'\ b$)
then show *?case*
using *lexpands-un-pwits-eq[OF 2(1,2,5)[unfolded P-def]]*
proof(*induction rule: lexpands-un.induct*)
case ($4\ s\ t1\ t2\ b$)

```

then have  $t1 \sqcup_s t2 \in \text{subterms } b$ 
  unfolding subterms-branch-def
  by (metis UN-iff UnI2 subterms-fm-simps(1) subterms-atom.simps(1) sub-
terms-refl)
with  $\langle \text{no-new-subterms } b \rangle$  have  $t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)$ 
  using no-new-subtermsD by blast
then have  $t1 \notin \text{Var } ' \text{wits } b$   $t2 \notin \text{Var } ' \text{wits } b$ 
  by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
with 4 show ?case
  by (auto simp: wits-subterms-def P-def subterms-branch-simps pwitsD(1))
next
case (5  $s$   $t1$   $t2$   $b$ )
then have  $t1 \sqcup_s t2 \in \text{subterms } b$ 
  unfolding subterms-branch-def
  by (metis UN-iff UnI2 subterms-fm-simps(1) subterms-atom.simps(1) sub-
terms-refl)
with  $\langle \text{no-new-subterms } b \rangle$  have  $t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)$ 
  using no-new-subtermsD by blast
then have  $t1 \notin \text{Var } ' \text{wits } b$   $t2 \notin \text{Var } ' \text{wits } b$ 
  by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
with 5 show ?case
  by (auto simp: wits-subterms-def P-def subterms-branch-simps pwitsD(1))
qed (auto simp: wits-subterms-def P-def)
next
case (3  $b'$   $b$ )
then show ?case
  using lexpands-int-pwits-eq[OF 3(1,2,5)[unfolded P-def]]
proof(induction rule: lexpands-int.induct)
  case (1  $s$   $t1$   $t2$   $b$ )
  then have  $t1 \sqcap_s t2 \in \text{subterms } b$ 
    unfolding subterms-branch-def
    by (metis UN-iff UnI2 subterms-fm-simps(1) subterms-atom.simps(1) sub-
terms-refl)
  with  $\langle \text{no-new-subterms } b \rangle$  have  $t1 \sqcap_s t2 \in \text{subterms } (\text{last } b)$ 
    using no-new-subtermsD by blast
  then have  $t1 \notin \text{Var } ' \text{wits } b$   $t2 \notin \text{Var } ' \text{wits } b$ 
    by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
  with 1 show ?case
    by (auto simp: wits-subterms-def P-def subterms-branch-simps pwitsD(1))
  next
  case (6  $s$   $t1$   $b$   $t2$ )
  then have  $t1 \notin \text{Var } ' \text{wits } b$   $t2 \notin \text{Var } ' \text{wits } b$ 
    by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
  with 6 show ?case
    by (auto simp: wits-subterms-def P-def subterms-branch-simps pwitsD(1))
  qed (auto simp: wits-subterms-def P-def)
next
case (4  $b'$   $b$ )
then show ?case

```

```

    using lexpands-diff-pwits-eq[OF 4(1,2,5)[unfolded P-def]]
  proof(induction rule: lexpands-diff.induct)
    case (1 s t1 t2 b)
    then have t1  $-_s$  t2  $\in$  subterms b
      unfolding subterms-branch-def
      by (metis UN-iff UnI2 subterms-fm-simps(1) subterms-atom.simps(1) sub-
terms-refl)
    with ⟨no-new-subterms b⟩ have t1  $-_s$  t2  $\in$  subterms (last b)
      using no-new-subtermsD by blast
    then have t1  $\notin$  Var ‘ wits b t2  $\notin$  Var ‘ wits b
      by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
    with 1 show ?case
      by (auto simp: wits-subterms-def P-def subterms-branch-simps dest: pwitsD(1))
  next
    case (4 s t1 t2 b)
    then have t1  $-_s$  t2  $\in$  subterms b
      unfolding subterms-branch-def
      by (metis AF-mem-subterms-branchD(2) subterms-branch-def)
    with ⟨no-new-subterms b⟩ have t1  $-_s$  t2  $\in$  subterms (last b)
      using no-new-subtermsD by blast
    then have t1  $\notin$  Var ‘ wits b t2  $\notin$  Var ‘ wits b
      by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
    with 4 show ?case
      by (auto simp: wits-subterms-def P-def subterms-branch-simps dest: pwitsD(1))
  qed (auto simp: wits-subterms-def P-def)
next
  case (5 b' b)
  then show ?case
  proof(induction rule: lexpands-single.induct)
    case (2 s t b)
    then have Single t  $\in$  subterms b
      by (auto intro: subterms-branch-subterms-atomI)
    with 2 have t  $\in$  subterms (last b)
      by (metis subterms-fmD(7) no-new-subtermsD(5))
    then have  $\forall c \in$  pwits b. Var c  $\neq$  t
      unfolding pwits-def wits-def
      using wits-def wits-subterms-last-disjnt by fastforce
    with ⟨t  $\in$  subterms (last b)⟩ show ?case
      using 2
      unfolding P-def
      by (auto simp: wits-subterms-last-disjnt[unfolded disjoint-iff] wits-subterms-def
subsetD
      dest: pwitsD(2))
  qed (auto simp: wits-subterms-def P-def)
next
  case (6 b' b)
  then have no-new-subterms (b' @ b) b' @ b  $\neq$  []
    using lexpands-no-new-subterms[OF lexpands.intros(6)] by blast+
  note subterms-branch-eq-if-no-new-subterms[OF this]

```

```

with 6 show ?case
proof(induction rule: lexpands-eq-induct')
  case (subst t1 t2 t1' t2' p l b)
  then have t1' ∈ subterms b
    using AT-eq-subterms-branchD by blast
  then show ?case unfolding P-def
proof(safe, goal-cases)
  case (1 c x)
  with subst have [simp]: p
    by (cases p) (simp add: P-def wits-subterms-def; blast)+
  from 1 subst have (Var c =s x) = subst-tlvl t1' t2' l
    by (simp add: P-def wits-subterms-def; blast)
  with 1 subst consider
    (refl) l = (t1' =s t1') t2' = Var c x = Var c
    | (t1'-left) l = (Var c =s t1') t2' = x
    | (t1'-right) l = (t1' =s x) t2' = Var c
  apply(cases (t1', t2', l) rule: subst-tlvl.cases)
  by (auto split: if-splits)
  then show ?case
    apply(cases)
    by (use 1 subst subterms-branch-eq-if-no-new-subterms in ⟨(simp add: P-def;
blast)⟩)
next
  case (2 c x)
  with subst have [simp]: p
    by (cases p) (simp add: P-def wits-subterms-def; blast)+
  from 2 subst have (x =s Var c) = subst-tlvl t1' t2' l
    by (simp add: P-def wits-subterms-def; blast)
  with 2 subst consider
    (refl) l = (t1' =s t1') t2' = Var c x = Var c
    | (t1'-left) l = (t1' =s Var c) t2' = x
    | (t1'-right) l = (x =s t1') t2' = Var c
  apply(cases (t1', t2', l) rule: subst-tlvl.cases)
  by (auto split: if-splits)
  then show ?case
    apply(cases)
    by (use 2 subst subterms-branch-eq-if-no-new-subterms in ⟨(simp add: P-def;
blast)⟩)
next
  case (3 c x)
  with subst have [simp]: p
    by (cases p) (simp add: P-def wits-subterms-def; blast)+
  from 3 subst have (x ∈s Var c) = subst-tlvl t1' t2' l
    by (simp add: P-def wits-subterms-def; blast)
  with 3 subst consider
    (refl) l = (t1' ∈s t1') t2' = Var c x = Var c
    | (t1'-left) l = (t1' ∈s Var c) t2' = x
    | (t1'-right) l = (x ∈s t1') t2' = Var c
  apply(cases (t1', t2', l) rule: subst-tlvl.cases)

```

```

      by (auto split: if-splits)
    then show ?case
      apply(cases)
      by (use 3 subst subterms-branch-eq-if-no-new-subterms in <(simp add: P-def;
blast)+>)
    qed
  next
    case (neq s t s' b)
    then show ?case
      using P-def by (simp add: wits-subterms-def) blast
    qed
  qed

```

lemma *lemma-2-bexpands*:

```

defines P ≡ (λ b c t. AT (Var c =s t) ∉ set b ∧ AT (t =s Var c) ∉ set b
  ∧ AT (t ∈s Var c) ∉ set b)
assumes bexpands bs' b b' ∈ bs' b ≠ []
assumes no-new-subterms b
assumes ∀ φ ∈ set b. ¬ is-literal φ → vars φ ∩ wits b = {}
assumes ∀ c ∈ pwits b. ∀ t ∈ wits-subterms b. P b c t
shows ∀ c ∈ pwits (b' @ b). ∀ t ∈ wits-subterms (b' @ b). P (b' @ b) c t
using assms(2-) bexpands-no-new-subterms[OF assms(2,4,3,5)]
proof(induction rule: bexpands.induct)
  case (1 bs' b)
  then show ?case
    using bexpands-nowit-wits-eq[OF 1(1-3)] bexpands-nowit-pwits-eq[OF 1(1-3,5)]
  proof(induction rule: bexpands-nowit.induct)
    case (1 p q b)
    then show ?case
      unfolding P-def wits-subterms-def
      by (fastforce dest: pwitsD)
  next
    case (2 p q b)
    then show ?case
      unfolding P-def wits-subterms-def
      by (fastforce dest: pwitsD)
  next
    case (3 s t1 t2 b)
    then have t1 ∉ Var ' wits b
      by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
    with 3 show ?case
      unfolding P-def wits-subterms-def
      by (fastforce simp: vars-branch-simps dest: pwitsD)
  next
    case (4 s t1 b t2)
    then have t2 ∉ Var ' wits b
      by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
    with 4 show ?case
      unfolding P-def wits-subterms-def

```

```

    by (fastforce simp: vars-branch-simps dest: pwitsD)
  next
  case (5 s t1 b t2)
  then have t2 ∉ Var ‘ wits b
    by (meson disjoint-iff wits-subterms-last-disjnt subterms-fmD)+
  with 5 show ?case
    unfolding P-def wits-subterms-def
    by (fastforce simp: vars-branch-simps dest: pwitsD)
qed
next
case (2 t1 t2 x bs b)
from bexpands-witD[OF 2(1)] have t1 ∉ Var ‘ wits b t2 ∉ Var ‘ wits b
  by (meson disjoint-iff-not-equal wits-subterms-last-disjnt)+
then have not-in-pwits: t1 ∉ Var ‘ pwits b t2 ∉ Var ‘ pwits b
  unfolding pwits-def by auto
with bexpands-witD[OF 2(1)] 2(2-) show ?case
  unfolding P-def wits-subterms-def
  unfolding bexpands-wit-pwits-eq[OF 2(1-3)] bexpands-wit-wits-eq[OF 2(1-3)]
  by safe (auto simp: vars-fm-vars-branchI[where ?x=x and ?b=b])
qed

```

lemma *subterms-branch-eq-if-wf-branch:*
 assumes *wf-branch b*
 shows *subterms-branch b = wits-subterms b*
proof –
 from *assms* obtain φ where *expandss b* [φ]
 unfolding *wf-branch-def* by blast
 then have *no-new-subterms* [φ]
 unfolding *no-new-subterms-def wits-def*
 by (*simp add: subterms-branch-simps*)
 with $\langle \text{expandss } b \text{ } [\varphi] \rangle$ have *no-new-subterms b*
 using *expandss-no-new-subterms* by blast
 with $\langle \text{expandss } b \text{ } [\varphi] \rangle$ *assms* show ?*thesis*
 by (*intro subterms-branch-eq-if-no-new-subterms*) *simp-all*
qed

lemma
 assumes *wf-branch b*
 shows *no-new-subterms-if-wf-branch: no-new-subterms b*
 and *no-wits-if-not-literal-if-wf-branch:*
 $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$
proof –
 from *assms* obtain φ where *expandss b* [φ]
 unfolding *wf-branch-def* by blast
 then have *no-new-subterms* [φ]
 by (*auto simp: no-new-subterms-def wits-def vars-branch-simps subterms-branch-simps*)
 from *expandss-no-new-subterms*[OF $\langle \text{expandss } b \text{ } [\varphi] \rangle$ - *this*] show *no-new-subterms*
b
 by *simp*

```

from expandss-no-wits-if-not-literal[OF  $\langle \text{expandss } b \text{ } [\varphi] \rangle$ ]
show  $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$ 
  unfolding wits-def by simp
qed

lemma lemma-2:
assumes wf-branch b
assumes  $c \in \text{pwits } b$ 
shows  $AT (\text{Var } c =_s t) \notin \text{set } b \wedge AT (t =_s \text{Var } c) \notin \text{set } b \wedge AT (t \in_s \text{Var } c) \notin \text{set } b$ 
proof -
from  $\langle \text{wf-branch } b \rangle$  obtain  $\varphi$  where expandss  $b \text{ } [\varphi]$ 
  using wf-branch-def by blast
have no-wits-if-not-literal:  $\forall \varphi \in \text{set } b'. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b' = \{\}$ 
  if expandss  $b' \text{ } [\varphi]$  for  $b'$ 
  using no-wits-if-not-literal-if-wf-branch that unfolding wf-branch-def by blast
have no-new-subterms: no-new-subterms  $b' \text{ } [\varphi]$  for  $b'$ 
  using no-new-subterms-if-wf-branch that unfolding wf-branch-def by blast
have  $AT (\text{Var } c =_s t) \notin \text{set } b \wedge AT (t =_s \text{Var } c) \notin \text{set } b \wedge AT (t \in_s \text{Var } c) \notin \text{set } b$ 
  using  $\langle \text{expandss } b \text{ } [\varphi] \rangle$  assms(2)
proof(induction  $b \text{ } [\varphi]$  arbitrary:  $c \text{ } t$  rule: expandss.induct)
  case 1
  then show ?case by simp
next
  case (2 b1 b2)
  note lemma-2-lexpands[OF this(1)] -
    no-new-subterms[OF this(3)] no-wits-if-not-literal[OF this(3)]
  with 2 show ?case
  using wf-branch-def wf-branch-not-Nil subterms-branch-eq-if-wf-branch
  by (metis AT-eq-subterms-branchD(1,2) AT-mem-subterms-branchD(1) expandss.intros(2))
next
  case (3 bs b2 b1)
  note lemma-2-bexpands[OF 3(1)] - -
    no-new-subterms[OF 3(3)] no-wits-if-not-literal[OF 3(3)]
  with 3 show ?case
  using wf-branch-def wf-branch-not-Nil subterms-branch-eq-if-wf-branch
  by (metis AT-eq-subterms-branchD(1,2) AT-mem-subterms-branchD(1) expandss.intros(3))
  qed
then show  $AT (\text{Var } c =_s t) \notin \text{set } b \wedge AT (t =_s \text{Var } c) \notin \text{set } b \wedge AT (t \in_s \text{Var } c) \notin \text{set } b$ 
  by safe
qed

```

Urelements

definition *urelems* $b \equiv \{x \in \text{subterms } b. \exists v. \forall \varphi \in \text{set } b. \text{urelem}' v \varphi x\}$

```

lemma finite-urelems: finite (urelems b)
proof –
  have urelems b  $\subseteq$  subterms b
    unfolding urelems-def urelem-def by blast
  with finite-subset finite-subterms-branch show ?thesis
    by blast
qed

lemma urelems-subsubterms: urelems b  $\subseteq$  subterms b
  unfolding urelems-def by blast

lemma is-Var-if-mem-urelems:  $t \in \text{urelems } b \implies \text{is-Var } t$ 
  unfolding urelems-def subterms-branch-def
  using is-Var-if-urelem' by auto

lemma urelems-subvars: urelems b  $\subseteq$  Var ‘ vars b
proof
  fix t assume t  $\in$  urelems b
  with urelems-subsubterms have t  $\in$  subterms b
    by blast
  moreover note is-Var-if-mem-urelems[OF  $\langle t \in \text{urelems } b \rangle$ ]
  then obtain x where t = Var x
    by (meson is-Var-def)
  ultimately have x  $\in$  vars b
    unfolding Un-vars-term-subterms-branch-eq-vars-branch[symmetric]
    by force
  with  $\langle t = \text{Var } x \rangle$  show t  $\in$  Var ‘ vars b
    by blast
qed

lemma types-term-inf:
  includes no member-ASCII-syntax
  assumes v1  $\vdash$  t : l1 v2  $\vdash$  t : l2
  shows inf v1 v2  $\vdash$  t : inf l1 l2
  using assms
  apply(induction t arbitrary: l1 l2)
    apply(auto simp: inf-min elim!: types-pset-term-cases
      intro: types-pset-term-intros' types-pset-term.intros(4–))
  done

lemma types-pset-atom-inf:
  fixes a :: 'a pset-atom
  assumes v1  $\vdash$  a v2  $\vdash$  a
  shows inf v1 v2  $\vdash$  a
  using assms
  by (auto simp: types-pset-atom.simps) (metis inf-min min-Suc-Suc types-term-inf)+

lemma types-pset-fm-inf:
  fixes  $\varphi$  :: 'a pset-fm

```

assumes $v1 \vdash \varphi \ v2 \vdash \varphi$
shows $\text{inf } v1 \ v2 \vdash \varphi$
using *assms types-pset-atom-inf*
unfolding *types-pset-fm-def* **by** *blast*

lemma *types-urelems*:

includes *no member-ASCII-syntax*
fixes $b :: 'a \text{ branch}$
assumes *wf-branch* $b \ v \vdash \text{last } b$
obtains v' **where** $\forall \varphi \in \text{set } b. v' \vdash \varphi \ \forall u \in \text{urelems } b. v' \vdash u : 0$

proof –

from *assms* **have** *expandss* $b \ [\text{last } b]$
unfolding *wf-branch-def* **by** *force*

define V **where** $V \equiv \{v. (\forall \varphi \in \text{set } b. v \vdash \varphi) \wedge (\forall x. x \notin \text{vars } b \longrightarrow v \ x = 0)\}$
have $V \neq \{\}$

proof –

from *types-expandss*[*OF* $\langle \text{expandss } b \ [\text{last } b] \rangle$, *simplified*] $\langle v \vdash \text{last } b \rangle$

obtain v **where** $v: \forall \varphi \in \text{set } b. v \vdash \varphi$

unfolding *vars-branch-simps* **by** *metis*

define v' **where** $v' \equiv \lambda x. \text{if } x \in \text{vars } b \text{ then } v \ x \text{ else } 0$

have $v' \vdash \varphi \longleftrightarrow v \vdash \varphi$ **if** $\varphi \in \text{set } b$ **for** $\varphi :: 'a \text{ pset-fm}$

apply(*intro types-pset-fm-if-on-vars-eq*)

using *that vars-fm-vars-branchI* **unfolding** v' -*def* **by** *metis*

with v **have** $\forall \varphi \in \text{set } b. v' \vdash \varphi$

by *blast*

then **have** $v' \in V$

unfolding V -*def* v' -*def* **by** *simp*

then **show** *?thesis*

by *blast*

qed

define $m\text{-}x$ **where** $m\text{-}x \equiv \lambda x. \text{ARG-MIN } (\lambda v. v \ x) \ v. v \in V$
with $\langle V \neq \{\} \rangle$ **have** $m\text{-}x: \forall v \in V. m\text{-}x \ x \leq v \ x \ m\text{-}x \ x \in V$ **for** x
using *arg-min-nat-le arg-min-natI* **by** (*metis ex-in-conv*)+

define M **where** $M \equiv \text{Finite-Set.fold inf } (\text{SOME } v. v \in V) (m\text{-}x \ ' \ \text{vars } b)$

note *finite-imageI*[**where** *?h=m-x*, *OF finite-vars-branch*[*of b*]]

note $M\text{-inf-eq} = \text{Inf-fin.eq-fold}[\text{symmetric}, \text{OF } \text{this}, \text{of } \text{SOME } v. v \in V]$

have $M\text{-leq}: M \ x \leq v \ x$ **if** $x \in \text{vars } b \ v \in V$ **for** $x \ v$

proof –

from *that* **have** $m\text{-}x \ x \in m\text{-}x \ ' \ \text{vars } b$

by *blast*

then **have** $M = \text{inf } (m\text{-}x \ x) (\prod_{\text{fin}} \text{insert } (\text{SOME } v. v \in V) (m\text{-}x \ ' \ \text{vars } b))$

unfolding M -*def* M -*inf-eq*

by (*simp add: Inf-fin.in-idem finite-vars-branch*)

with $m\text{-}x$ **that** **show** *?thesis*

by (*simp add: inf.coboundedI1*)

qed

```

moreover have  $M \in V$ 
  unfolding  $M$ -def  $M$ -inf-eq using finite-vars-branch[of  $b$ ]
proof(induction rule: finite-induct)
  case empty
  with  $\langle V \neq \{\} \rangle$  show ?case
    by (simp add: some-in-eq)
next
  case (insert x F)
  then have  $M'$ -eq:  $\prod_{fin} \text{insert } (SOME\ v.\ v \in V)\ (m\text{-}x\ ' \text{insert } x\ F)$ 
    =  $\text{inf } (m\text{-}x\ x)\ (\prod_{fin} \text{insert } (SOME\ v.\ v \in V)\ (m\text{-}x\ ' F))$  (is - = ? $M'$ )
    by (simp add: insert-commute)
  from types-pset-fm-inf insert have  $\forall \varphi \in \text{set } b.\ ?M' \vdash \varphi$ 
    using  $V$ -def  $m$ -x(2) by blast
  moreover have (inf w v)  $x = 0$  if  $x \notin \text{vars } b$   $w \in V$   $v \in V$  for  $w\ v$ 
    using that by (simp add: V-def)
  with insert.IH m-x(2) have  $\forall x.\ x \notin \text{vars } b \longrightarrow ?M' x = 0$ 
    by (simp add: V-def)
  ultimately have ? $M' \in V$ 
    using  $V$ -def by blast
  with  $M'$ -eq show ?case
    by metis
qed
moreover have  $M \vdash u : 0$  if  $u \in \text{urelems } b$  for  $u$ 
proof -
  from that obtain  $v$  where  $v:\ \forall \varphi \in \text{set } b.\ \text{urelem}'\ v\ \varphi\ u$ 
    unfolding urelems-def by blast
  define  $v'$  where  $v' \equiv \lambda x.\ \text{if } x \in \text{vars } b\ \text{then } v\ x\ \text{else } 0$ 
  have  $v' \vdash \varphi \longleftrightarrow v \vdash \varphi$  if  $\varphi \in \text{set } b$  for  $\varphi :: 'a\ \text{pset-fm}$ 
    apply(intro types-pset-fm-if-on-vars-eq)
    using that vars-fm-vars-branchI unfolding  $v'$ -def by metis
  with  $v$  have  $\forall \varphi \in \text{set } b.\ v' \vdash \varphi$ 
    by blast
  then have  $v' \in V$ 
    unfolding  $V$ -def  $v'$ -def by auto
  moreover obtain  $uv$  where  $uv: u = \text{Var } uv$ 
    using  $v$  is-Var-if-urelem' wf-branch-not-Nil[OF  $\langle \text{wf-branch } b \rangle$ ]
    by (metis is-Var-def last-in-set)
  then have  $v' \vdash u : 0$ 
    using  $v$  wf-branch-not-Nil[OF  $\langle \text{wf-branch } b \rangle$ , THEN last-in-set]
    unfolding  $v'$ -def
    by (auto elim!: types-pset-term-cases(2) intro!: types-pset-term-intros'(2))
  ultimately show  $M \vdash u : 0$ 
    using  $M$ -leq[where ? $v=v'$ ]  $\langle M \in V \rangle$ [unfolded V-def] unfolding  $uv$ 
    by (fastforce elim!: types-pset-term-cases(2) intro!: types-pset-term-intros'(2))
qed
ultimately show ?thesis
  using that unfolding  $V$ -def by auto
qed

```

lemma *mem-urelems-if-urelem-last*:
assumes *wf-branch b*
assumes *urelem (last b) x x ∈ subterms (last b)*
shows *x ∈ urelems b*
proof –
from *assms* **have** *x ∈ subterms b*
unfolding *subterms-branch-def* **by** *auto*
moreover note *urelem-invar-if-wf-branch[OF assms]*
ultimately show *?thesis*
unfolding *urelems-def urelem-def* **by** *blast*
qed

lemma *not-urelem-comps-if-compound*:
assumes *f t1 t2 ∈ subterms b f ∈ {(Π_s), (\sqcup_s), ($-_s$)}*
shows *t1 \notin urelems b t2 \notin urelems b*
proof –
from *assms* **obtain** φ **where** *$\varphi \in \text{set } b f t1 t2 \in \text{subterms } \varphi$*
unfolding *subterms-branch-def* **by** *auto*
note *not-urelem-comps-if-compound[of f t1 t2, OF this(2) assms(2)]*
with $\langle \varphi \in \text{set } b \rangle$ **show** *t1 \notin urelems b t2 \notin urelems b*
unfolding *urelems-def urelem-def* **by** *auto*
qed

Realization of an Open Branch

definition *base-vars b* \equiv *Var ‘ pwits b \cup urelems b*

lemma *finite-base-vars: finite (base-vars b)*
unfolding *base-vars-def finite-Un*
using *finite-pwits[THEN finite-imageI] finite-urelems* **by** *blast*

lemma *pwits-sub-base-vars*:
shows *Var ‘ pwits b \subseteq base-vars b*
unfolding *base-vars-def*
by *blast*

lemma *base-vars-sub-vars: base-vars b \subseteq Var ‘ vars b*
unfolding *base-vars-def pwits-def wits-def*
using *urelems-sub-vars* **by** *blast*

definition *subterms'* $::$ *'a branch \Rightarrow 'a pset-term set* **where**
subterms' b \equiv subterms b – base-vars b

definition *bgraph* $::$ *'a branch \Rightarrow ('a pset-term, 'a pset-term \times 'a pset-term)*
pre-digraph **where**
bgraph b \equiv
let
vs = base-vars b \cup subterms' b
in

```

(| verts = vs,
 arcs = {(s, t). AT (s ∈s t) ∈ set b},
 tail = fst,
 head = snd |)

```

lemma *base-vars-Un-subterms'-eq-subterms:*

base-vars b ∪ subterms' b = subterms b

unfolding *subterms'-def*

using *base-vars-subs-vars vars-branch-subs-subterms-branch* **by** *fastforce*

lemma *finite-base-vars-Un-subterms': finite (base-vars b ∪ subterms' b)*

unfolding *base-vars-Un-subterms'-eq-subterms*

using *finite-subterms-branch* .

lemma *verts-bgraph: verts (bgraph b) = base-vars b ∪ subterms' b*

unfolding *bgraph-def Let-def* **by** *simp*

lemma *verts-bgraph-eq-subterms: verts (bgraph b) = subterms b*

unfolding *verts-bgraph base-vars-Un-subterms'-eq-subterms ..*

lemma *finite-subterms': finite (subterms' b)*

unfolding *subterms'-def* **using** *finite-base-vars finite-subterms-branch*

by *auto*

lemma *base-vars-subterms'-disjnt: base-vars b ∩ subterms' b = {}*

unfolding *subterms'-def* **by** *fastforce*

lemma *wits-subterms-eq-base-vars-Un-subterms':*

assumes *wf-branch b*

shows *wits-subterms b = base-vars b ∪ subterms' b*

unfolding *subterms-branch-eq-if-wf-branch[OF assms, symmetric]* *subterms'-def*

using *base-vars-subs-vars vars-branch-subs-subterms-branch*

by *fastforce*

lemma *in-subterms'-if-AT-mem-in-branch:*

assumes *wf-branch b*

assumes *AT (s ∈_s t) ∈ set b*

shows *s ∈ base-vars b ∪ subterms' b*

and *t ∈ base-vars b ∪ subterms' b*

using *assms*

using *wits-subterms-eq-base-vars-Un-subterms' AT-mem-subterms-branchD*

using *subterms-branch-eq-if-wf-branch*

by *blast+*

lemma *in-subterms'-if-AF-mem-in-branch:*

assumes *wf-branch b*

assumes *AF (s ∈_s t) ∈ set b*

shows *s ∈ base-vars b ∪ subterms' b*

and *t ∈ base-vars b ∪ subterms' b*

```

using assms
using wits-subterms-eq-base-vars-Un-subterms' AF-mem-subterms-branchD
using subterms-branch-eq-if-wf-branch
by blast+

lemma in-subterms'-if-AT-eq-in-branch:
assumes wf-branch b
assumes AT (s =s t) ∈ set b
shows s ∈ base-vars b ∪ subterms' b
  and t ∈ base-vars b ∪ subterms' b
using assms
using wits-subterms-eq-base-vars-Un-subterms' AT-eq-subterms-branchD
using subterms-branch-eq-if-wf-branch
by blast+

lemma in-subterms'-if-AF-eq-in-branch:
assumes wf-branch b
assumes AF (s =s t) ∈ set b
shows s ∈ base-vars b ∪ subterms' b
  and t ∈ base-vars b ∪ subterms' b
using assms
using wits-subterms-eq-base-vars-Un-subterms' AF-eq-subterms-branchD
using subterms-branch-eq-if-wf-branch
by blast+

lemma mem-subterms-fm-last-if-mem-subterms-branch:
assumes wf-branch b
assumes t ∈ subterms b ¬ is-Var t
shows t ∈ subterms (last b)
using assms
unfolding subterms-branch-eq-if-wf-branch[OF ‹wf-branch b›]
unfolding subterms'-def wits-subterms-def by force

locale open-branch =
  fixes b :: 'a branch
  assumes wf-branch: wf-branch b and bopen: bopen b and types: ∃ v. v ⊢ last b
  and infinite-vars: infinite (UNIV :: 'a set)
begin

sublocale fin-digraph-bgraph: fin-digraph bgraph b
proof
  show finite (verts (bgraph b))
    using finite-base-vars finite-subterms'
    by (auto simp: bgraph-def Let-def)

  show finite (arcs (bgraph b))
    using [[simproc add: finite-Collect]]
    by (auto simp: bgraph-def Let-def inj-on-def intro!: finite-vimageI)

```

qed (use in-subterms'-if-AT-mem-in-branch[OF wf-branch]
in $\langle(\text{fastforce simp: bgraph-def Let-def})+\rangle$)

lemma member-seq-if-cas:
fin-digraph-bgraph.cas t1 is t2
 \implies member-seq t1 (map ($\lambda e.$ tail (bgraph b) e \in_s head (bgraph b) e) is) t2
by (induction is arbitrary: t1 t2) auto

lemma member-cycle-if-cycle:
fin-digraph-bgraph.cycle c
 \implies member-cycle (map ($\lambda e.$ tail (bgraph b) e \in_s head (bgraph b) e) c)
unfolding pre-digraph.cycle-def
by (cases c) (auto simp: member-seq-if-cas)

sublocale dag-bgraph: dag bgraph b

proof(unfold-locales, goal-cases)

case (1 e)

show ?case

proof

assume tail (bgraph b) e = head (bgraph b) e

then obtain t **where** AT (t \in_s t) \in set b

using 1 **unfolding** bgraph-def Let-def **by** auto

then have member-cycle [(t \in_s t)] (t \in_s t) \in Atoms (set b)

by (auto simp: Atoms-def)

then have bclosed b

using memberCycle **by** (metis empty-iff empty-set set-ConsD subsetI)

with bopen **show** False

by blast

qed

next

case (2 e1 e2)

then show ?case

by (auto simp: bgraph-def Let-def arc-to-ends-def)

next

case 3

show ?case

proof

assume $\exists c.$ *fin-digraph-bgraph.cycle c*

then obtain c **where** *fin-digraph-bgraph.cycle c*

by blast

then have member-cycle (map ($\lambda e.$ (tail (bgraph b) e \in_s head (bgraph b) e)) c)

using member-cycle-if-cycle **by** blast

moreover

from \langle *fin-digraph-bgraph.cycle c* \rangle **have** set c \subseteq arcs (bgraph b)

by (meson *fin-digraph-bgraph.cycle-def pre-digraph.awalk-def*)

then have set (map ($\lambda e.$ (tail (bgraph b) e \in_s head (bgraph b) e)) c) \subseteq Atoms

(set b)

unfolding bgraph-def Let-def Atoms-def **by** auto

ultimately have bclosed b

using *memberCycle* **by** *blast*
with *bopen* **show** *False* **by** *blast*
qed
qed

definition $I :: 'a \text{ pset-term} \Rightarrow \text{hf}$ **where**
 $I \equiv \text{SOME } f. \text{inj-on } f \text{ (subterms } b)$
 $\wedge (\forall p. \text{hcard } (f \ p) > 2 * \text{card } (\text{base-vars } b \cup \text{subterms}' \ b))$

lemma (**in** $-$) *Ex-set-family*:
assumes *finite P*
shows $\exists I. \text{inj-on } I \ P \wedge (\forall p. \text{hcard } (I \ p) \geq n)$
proof $-$
from $\langle \text{finite } P \rangle$ **obtain** ip **where** $ip: \text{bij-betw } ip \ P \ \{.. < \text{card } P\}$
using *to-nat-on-finite* **by** *blast*
let $?I = \text{ord-of } o \ ((+) \ n) \ o \ ip$
from ip **have** $\text{inj-on } ?I \ P$
by (*auto simp: inj-on-def bij-betw-def*)
moreover **have** $\text{hcard } (?I \ p) \geq n$ **for** p
by *simp*
ultimately **show** $?thesis$
by *auto*
qed

lemma
shows $\text{inj-on-}I: \text{inj-on } I \text{ (subterms } b)$
and $\text{card-}I: \text{hcard } (I \ p) > 2 * \text{card } (\text{base-vars } b \cup \text{subterms}' \ b)$
proof $-$
have $\exists f. \text{inj-on } f \text{ (subterms } b)$
 $\wedge (\forall p. \text{hcard } (f \ p) > 2 * \text{card } (\text{base-vars } b \cup \text{subterms}' \ b))$
using *Ex-set-family finite-subterms-branch* **by** (*metis less-eq-Suc-le*)
from *someI-ex[OF this]*
show $\text{inj-on } I \text{ (subterms } b)$
 $\text{hcard } (I \ p) > 2 * \text{card } (\text{base-vars } b \cup \text{subterms}' \ b)$
unfolding *I-def* **by** *blast+*
qed

lemma
shows $\text{inj-on-base-vars-}I: \text{inj-on } I \text{ (base-vars } b)$
and $\text{inj-on-subterms}'\text{-}I: \text{inj-on } I \text{ (subterms}' \ b)$
proof $-$
from *base-vars-Un-subterms'-eq-subterms* **have**
 $\text{base-vars } b \subseteq \text{subterms } b \ \text{subterms}' \ b \subseteq \text{subterms } b$
using *wf-branch-not-Nil[OF wf-branch]* **by** *blast+*
with *inj-on-I* **show** $\text{inj-on } I \text{ (base-vars } b) \ \text{inj-on } I \text{ (subterms}' \ b)$
unfolding *inj-on-def* **by** *blast+*
qed

definition $eq \equiv \text{symcl} (\{(s, t). AT (s =_s t) \in \text{set } b\})^=$

lemma *refl-eq*: *refl eq*

unfolding *eq-def symcl-def refl-on-def* **by** *auto*

lemma *trans-eq*:

assumes *lin-sat b* **shows** *trans eq*

proof

fix $s\ t\ u$ **assume** *assms*: $(s, t) \in eq\ (t, u) \in eq$

have $(s, u) \in eq$ **if** $s \neq t\ t \neq u$

proof –

from *that assms* **have**

$s-t$: $AT (s =_s t) \in \text{set } b \vee AT (t =_s s) \in \text{set } b$ **and**

$t-u$: $AT (t =_s u) \in \text{set } b \vee AT (u =_s t) \in \text{set } b$

unfolding *eq-def symcl-def* **by** *fastforce+*

note *intros* = *lexpands-eq.intros(1,3)*[

*THEN lexpands.intros(6), THEN $\langle \text{lin-sat } b \rangle$ [*THEN lin-satD*]*

note *intros'* = *intros*[**where** $?x=AT (s =_s u)$] *intros*[**where** $?x=AT (u =_s s)$]

from $s-t\ t-u$ **that** **have** $AT (s =_s u) \in \text{set } b \vee AT (u =_s s) \in \text{set } b$

by *safe (simp-all add: intros')*

then **show** *?thesis*

unfolding *eq-def symcl-def* **by** *auto*

qed

with *assms* **show** $(s, u) \in eq$

by (*cases* $s \neq t \wedge t \neq u$) (*auto simp: eq-def*)

qed

lemma *sym-eq*: *sym eq*

unfolding *eq-def symcl-def sym-def* **by** *auto*

lemma *equiv-eq*: *lin-sat b \implies equiv UNIV eq*

by (*rule equivI*) (*use refl-eq trans-eq sym-eq UNIV-I in safe*)

lemma *not-dominated-if-pwits*:

assumes $x \in \text{Var}$ ‘*pwits b*’ **shows** $\neg s \rightarrow_{\text{bgraph } b} x$

proof –

from *assms* **obtain** x' **where** $x = \text{Var } x'\ x' \in \text{pwits } b$

by *blast*

from *lemma-2(3)*[*OF wf-branch this(2)*] *this(1)* **show** $\neg s \rightarrow_{\text{bgraph } b} x$

unfolding *arcs-ends-def arc-to-ends-def* **by** (*auto simp: bgraph-def*)

qed

lemma *parents-empty-if-pwits*:

assumes $x \in \text{Var}$ ‘*pwits b*’ **shows** *parents (bgraph b) x* = $\{\}$

using *not-dominated-if-pwits*[*OF assms*] **unfolding** *bgraph-def* **by** *simp*

lemma *not-AT-mem-if-urelem*:

assumes $t \in \text{urelems } b$

shows $AT (s \in_s t) \notin \text{set } b$

proof
assume $AT (s \in_s t) \in \text{set } b$
with $\text{assms urelem-invar-if-wf-branch}[OF \text{ wf-branch}]$ **have** $\text{urelem } (AT (s \in_s t))$
 t
by $(\text{meson types types-urelems urelem-def wf-branch})$
then show False
unfolding urelem-def
by $(\text{auto dest!: types-fmD simp: types-pset-atom.simps dest: types-term-unique})$
qed

lemma $\text{not-dominated-if-urelems}$:
assumes $t \in \text{urelems } b$
shows $\neg s \rightarrow_{\text{bgraph } b} t$
using $\text{not-AT-mem-if-urelem}[OF \text{ assms}]$ **unfolding** bgraph-def **by** auto

lemma $\text{parents-empty-if-urelems}$:
assumes $t \in \text{urelems } b$
shows $\text{parents } (\text{bgraph } b) t = \{\}$
using $\text{not-dominated-if-urelems}[OF \text{ assms}]$ **by** simp

lemma $\text{not-dominated-if-base-vars}$:
assumes $x \in \text{base-vars } b$
shows $\neg s \rightarrow_{\text{bgraph } b} x$
using $\text{assms not-dominated-if-urelems not-dominated-if-pwits}$
unfolding base-vars-def **by** blast

lemma $\text{parents-empty-if-base-vars}$:
assumes $x \in \text{base-vars } b$
shows $\text{parents } (\text{bgraph } b) x = \{\}$
using $\text{not-dominated-if-base-vars}[OF \text{ assms}]$ **by** blast

lemma $\text{eq-class-sub-subterms}$: $\text{eq } \{t\} \subseteq \{t\} \cup \text{subterms } b$
proof –
have $\text{eq } - \text{Id} \subseteq \text{subterms } b \times \text{subterms } b$
by $(\text{auto simp: AT-eq-subterms-branchD eq-def symcl-def})$
then show $\text{eq } \{t\} \subseteq \{t\} \cup \text{subterms } b$
by blast
qed

lemma finite-eq-class : $\text{finite } (\text{eq } \{x\})$
using $\text{eq-class-sub-subterms finite-subterms-branch}$
using finite-subset **by** fastforce

lemma $\text{finite-I-image-eq-class}$: $\text{finite } (I \text{ ' eq } \{x\})$
using finite-eq-class **by** blast

context
begin

```

interpretation realisation bgraph b base-vars b subterms' b I eq
proof(unfold-locales)
  from base-vars-subterms'-disjnt show  $\text{base-vars } b \cap \text{subterms}' b = \{\}$  .

  show  $\text{verts } (\text{bgraph } b) = \text{base-vars } b \cup \text{subterms}' b$ 
    unfolding bgraph-def by simp

  from not-dominated-if-base-vars show  $\bigwedge p t. p \in \text{base-vars } b \implies \neg t \rightarrow \text{bgraph } b$ 
  p .
qed

lemmas realisation = realisation-axioms

lemma card-realisation:
   $\text{hcard } (\text{realise } t) \leq 2 * \text{card } (\text{subterms } b)$ 
proof(induction t rule: realise.induct)
  case (1 x)
  then have  $\text{hcard } (\text{realise } x) = \text{card } (\text{realise } ' \text{parents } (\text{bgraph } b) x \cup I ' \text{eq-class } x)$ 
  using hcard-HF Zero-hf-def parents-empty-if-base-vars
  using finite-I-image-eq-class by force
  also have  $\dots \leq \text{card } (\text{realise } ' \text{parents } (\text{bgraph } b) x) + \text{card } (I ' \text{eq-class } x)$ 
  using card-Un-le by blast
  also have  $\dots \leq \text{card } (\text{parents } (\text{bgraph } b) x) + \text{card } (\text{eq-class } x)$ 
  using card-image-le[OF fin-digraph-bgraph.finite-parents]
  using card-image-le[OF finite-eq-class]
  by (metis add-le-mono)
  also have  $\dots \leq \text{card } (\text{subterms } b) + \text{card } (\text{eq-class } x)$ 
  using fin-digraph-bgraph.parents-subs-verts[unfolded verts-bgraph-eq-subterms]
  using card-mono[OF finite-subterms-branch]
  by (simp add: 1.hyps not-dominated-if-base-vars)
  also have  $\dots \leq \text{card } (\text{subterms } b) + \text{card } (\{x\} \cup \text{subterms } b)$ 
  apply (intro add-le-mono card-mono[where ?B={x} ∪ subterms b])
  using eq-class-subs-subterms finite-subterms-branch by auto
  also have  $\dots \leq 2 * \text{card } (\text{subterms } b)$ 
proof –
  from 1 have  $x \in \text{subterms } b$ 
  using 1.prem1 verts-bgraph verts-bgraph-eq-subterms wf-branch-not-Nil[OF wf-branch]
  by blast
  then show ?thesis
  unfolding mult-2 by (metis insert-absorb insert-is-Un order-refl)
qed
finally show ?case .
next
  case (2 t)
  then have  $\text{hcard } (\text{realise } t) = \text{card } (\text{realise } ' \text{parents } (\text{bgraph } b) t)$ 
  using hcard-HF[OF finite-realisation-parents] by simp
  also have  $\dots \leq \text{card } (\text{parents } (\text{bgraph } b) t)$ 

```

```

    using card-image-le by blast
  also have ... ≤ card (subterms b)
    using fin-digraph-bgraph.parents-subs-verts wf-branch-not-Nil[OF wf-branch]
    unfolding verts-bgraph-eq-subterms
    by (metis card-mono fin-digraph-bgraph.finite-verts verts-bgraph-eq-subterms)
  finally show ?case
    unfolding base-vars-Un-subterms'-eq-subterms by auto
next
  case (3 t)
  then show ?case by simp
qed

lemma I-neq-realise: I x ≠ realise y
proof -
  note card-realisation[of y]
  moreover have hcard (I x) > 2 * card (subterms b)
    using card-I wf-branch
    by (simp add: subterms-branch-eq-if-wf-branch wits-subterms-eq-base-vars-Un-subterms')
  ultimately show ?thesis
    by (metis linorder-not-le)
qed

end

sublocale realisation bgraph b base-vars b subterms' b I eq
  rewrites (∧ x y. I x ≠ realise y) ≡ Trueprop True
    and ∧ P. (True ⇒ P) ≡ Trueprop P
    and ∧ P Q. (True ⇒ PROP P ⇒ PROP Q) ≡ (PROP P ⇒ True ⇒
PROP Q)
  using realisation I-neq-realise by simp-all

lemma eq-class-singleton-if-pwits:
  assumes x ∈ Var ' pwits b shows eq-class x = {x}
proof -
  from assms obtain x' where x = Var x' x' ∈ pwits b
    by blast
  have False if eq-class x ≠ {x}
  proof -
    have x ∈ eq-class x
      by (simp add: eq-def symcl-def)
    with that obtain y where y ∈ eq-class x y ≠ x
      by auto
    then have AT (y =s x) ∈ set b ∨ AT (x =s y) ∈ set b
      unfolding eq-def symcl-def by auto
    with lemma-2(1,2)[OF wf-branch ⟨x' ∈ pwits b⟩ ⟨x = Var x'⟩ show False
      by blast
  qed
  with assms show ?thesis by blast
qed

```

lemma *realise-pwits*:

$x \in \text{Var} \text{ 'pwits } b \implies \text{realise } x = \text{HF } \{I x\}$

unfolding *realise.simps(1)[OF pwits-subst-base-vars[THEN subsetD]]*

by (*auto simp: eq-class-singleton-if-pwits parents-empty-if-pwits*)

lemma *I-in-realise-if-base-vars[simp]*:

$s \in \text{base-vars } b \implies I s \in \text{realise } s$

using *refl-eq by (simp add: finite-I-image-eq-class refl-on-def)*

lemma *realise-neq-if-base-vars-and-subterms'*:

assumes $s \in \text{base-vars } b \ t \in \text{subterms}' b$

shows $\text{realise } s \neq \text{realise } t$

proof –

from *assms* **have** $I s \notin \text{realise } t$

using *I-neq-realise* **by** *auto*

with *I-in-realise-if-base-vars* *assms(1)* **show** *?thesis*

by *metis*

qed

lemma *AT-mem-branch-wits-subtermsD*:

assumes $AT (s \in_s t) \in \text{set } b$

shows $s \in \text{wits-subterms } b \ t \in \text{wits-subterms } b$

using *assms AT-mem-subterms-branchD subterms-branch-eq-if-wf-branch wf-branch*

by *blast+*

lemma *AF-mem-branch-wits-subtermsD*:

assumes $AF (s \in_s t) \in \text{set } b$

shows $s \in \text{wits-subterms } b \ t \in \text{wits-subterms } b$

using *assms AF-mem-subterms-branchD subterms-branch-eq-if-wf-branch wf-branch*

by *blast+*

lemma *AT-eq-branch-wits-subtermsD*:

assumes $AT (s =_s t) \in \text{set } b$

shows $s \in \text{wits-subterms } b \ t \in \text{wits-subterms } b$

using *assms AT-eq-subterms-branchD subterms-branch-eq-if-wf-branch wf-branch*

by *blast+*

lemma *AF-eq-branch-wits-subtermsD*:

assumes $AF (s =_s t) \in \text{set } b$

shows $s \in \text{wits-subterms } b \ t \in \text{wits-subterms } b$

using *assms AF-eq-subterms-branchD subterms-branch-eq-if-wf-branch wf-branch*

by *blast+*

lemma *realisation-if-AT-mem*:

assumes $AT (s \in_s t) \in \text{set } b$

shows $\text{realise } s \in \text{realise } t$

proof –

from *assms* **have** $t \in \text{base-vars } b \cup \text{subterms}' b$

using *in-subterms'-if-AT-mem-in-branch(2)* *wf-branch* **by** *blast*
 moreover from *assms* have $s \rightarrow_{\text{bgraph}} b \ t$
 unfolding *arcs-ends-def* *arc-to-ends-def* **by** (*simp add: bgraph-def*)
 ultimately show *?thesis*
 by (*cases t rule: realise.cases*) *auto*
qed

lemma *AT-eq-urelems-subterms'-cases:*
 includes *no member-ASCII-syntax*
 assumes $AT \ (s =_s \ t) \in \text{set } b$
 obtains $(\text{urelems}) \ s \in \text{urelems } b \ t \in \text{urelems } b \mid$
 $(\text{subterms}') \ s \in \text{subterms}' \ b \ t \in \text{subterms}' \ b$

proof –
 from *types* obtain *v* where $v \vdash \text{last } b$
 by *blast*
 with *types-urelems wf-branch* obtain *v'*
 where $v': \forall \varphi \in \text{set } b. \ v' \vdash \varphi \ \forall u \in \text{urelems } b. \ v' \vdash u : 0$
 by *blast*
 with *assms* have $v' \vdash AT \ (s =_s \ t)$
 by *blast*
 then obtain *lst* where $lst: v' \vdash s : \text{lst } v' \vdash t : \text{lst}$
 by (*auto dest!: types-fmD(6) simp: types-pset-atom.simps*)
 note *mem-subterms = AT-eq-subterms-branchD[OF assms]*
 with *v'* have $t \in \text{urelems } b$ if $s \in \text{urelems } b$
 using *that lst types-term-unique urelems-def* **by** *fastforce*
 moreover from *assms* have $s \notin \text{Var } \langle \text{pwits } b \ t \notin \text{Var } \langle \text{pwits } b \rangle \rangle$
 using *lemma-2(1,2)[OF wf-branch]* **by** *blast+*
 moreover have $t \in \text{subterms}' \ b$ if $s \in \text{subterms}' \ b$
proof –
 have $s \notin \text{urelems } b$
 using *that B-T-partition-verts(1) unfolding base-vars-def* **by** *blast*
 with $v'(1)$ *mem-subterms(1)* have $\neg v' \vdash s : 0$
 using *urelems-def* **by** *blast*
 with $lst \ v'(2)$ have $t \notin \text{urelems } b$
 using *types-term-unique* **by** *metis*
 with $\langle t \notin \text{Var } \langle \text{pwits } b \rangle \rangle \langle t \in \text{subterms } b \rangle$ **show** $t \in \text{subterms}' \ b$
 by (*simp add: base-vars-def subterms'-def*)
qed
 ultimately show *?thesis*
 using *that mem-subterms*
 by (*cases s \in \text{subterms}' \ b*) (*auto simp: base-vars-def subterms'-def*)
qed

lemma *realisation-if-AT-eq:*
 assumes *lin-sat b*
 assumes $AT \ (s =_s \ t) \in \text{set } b$
 shows *realise s = realise t*
proof –
 from *assms(2)* **show** *?thesis*

proof(*cases rule: AT-eq-urelems-subterms'-cases*)
case *urelems*
then have $s \in \text{base-vars } b \ t \in \text{base-vars } b$
by (*simp-all add: base-vars-def*)
moreover from *assms* **have** $(s, t) \in \text{eq}$
unfolding *eq-def symcl-def* **by** *blast*
then have $I \text{ ' eq-class } s = I \text{ ' eq-class } t$
using *equiv-eq[OF assms(1)]* **by** (*simp add: equiv-class-eq-iff*)
ultimately show *?thesis*
using *urelems* **by** (*simp add: parents-empty-if-urelems*)
next
case *subterms'*
have *False* **if** $\text{realise } s \neq \text{realise } t$
proof –
from *that* **have** $\text{hfset } (\text{realise } s) \neq \text{hfset } (\text{realise } t)$
by (*metis HF-hfset*)
then obtain $a \ s' \ t'$ **where**
 $a \in \text{realise ' parents } (\text{bgraph } b) \ s'$
 $a \notin \text{realise ' parents } (\text{bgraph } b) \ t'$
and $s'-t': s' = s \wedge t' = t \vee s' = t \wedge t' = s$
using *subterms'* **by** *auto blast+*
with *subterms'* **have** $s' \in \text{subterms'} \ b \ t' \in \text{subterms'} \ b$
by *auto*
with a **obtain** u **where** $u: a = \text{realise } u \ u \rightarrow_{\text{bgraph } b} s'$
using *subterms' dominates-if-mem-realisation* **by** *auto*
then have $u \neq s'$
using *dag-bgraph.adj-not-same* **by** *blast*
from u **have** $AT \ (u \in_s \ s') \in \text{set } b$
unfolding *bgraph-def Let-def* **using** *dag-bgraph.adj-not-same* **by** *auto*
note *lexpands-eq.intros(1,3)[OF assms(2) this, THEN lexpands.intros(6)]*
with $\langle \text{lin-sat } b \rangle \ s'-t' \ \langle u \neq s' \rangle$ **have** $AT \ (u \in_s \ t') \in \text{set } b$
unfolding *lin-sat-def* **by** (*auto split: if-splits*)
from *realisation-if-AT-mem[OF this]* $\langle a = \text{realise } u \rangle$ **have** $a \in \text{realise } t'$
by *blast*
with a **show** *False*
using $\langle t' \in \text{subterms'} \ b \rangle$ **by** *simp*
qed
then show *?thesis* **by** *blast*
qed
qed

lemma *realise-base-vars-if-AF-eq*:
assumes *sat b*
assumes $AF \ (x =_s \ y) \in \text{set } b$
assumes $x \in \text{base-vars } b \ \vee \ y \in \text{base-vars } b$
shows $\text{realise } x \neq \text{realise } y$
proof(*cases x \in base-vars b \wedge y \in base-vars b*)
case *False*
with *assms(3)* **show** *?thesis*

```

    using realise-neq-if-base-vars-and-subterms' I-in-realise-if-base-vars
    by (metis hempty-iff realise.elims)
next
case True
from assms bopen have  $x \neq y$ 
  using neqSelf by blast
moreover from assms bopen have  $AT (x =_s y) \notin set\ b$ 
  using contr by blast
moreover have  $AT (y =_s x) \notin set\ b$ 
proof
  assume  $AT (y =_s x) \in set\ b$ 
  note lexpands-eq.intros(2)[OF this assms(2), THEN lexpands.intros(6)]
  with <sat b>[THEN satD(1), THEN lin-satD] have  $AF (x =_s x) \in set\ b$ 
    by auto
  with bopen neqSelf show False
    by blast
qed
ultimately have  $(x, y) \notin eq$ 
  unfolding eq-def symcl-def by auto

then have  $x \notin eq\text{-class}\ y$ 
  by (meson Image-singleton-iff symE sym-eq)
then have  $I\ x \notin I\ ' eq\text{-class}\ y$ 
  using inj-on-I AF-eq-subterms-branchD[OF assms(2)]
  using eq-class-subs-subterms inj-onD by fastforce
then have  $I\ ' eq\text{-class}\ x \neq I\ ' eq\text{-class}\ y$ 
  using refl-eq
  by (metis Image-singleton-iff UNIV-I imageI refl-onD)

with <x ∈ base-vars b ∧ y ∈ base-vars b> show realise  $x \neq realise\ y$ 
  using hunion-hempty-left[unfolded Zero-hf-def]
  using inj-on-HF[THEN inj-onD] finite-I-image-eq-class
  by (force simp: parents-empty-if-base-vars)
qed

lemma Ex-AT-eq-mem-subterms-last-if-subterms':
  assumes  $t \in subterms'\ b$ 
  obtains  $t \in subterms (last\ b) - base\text{-vars}\ b$ 
  |  $t'$  where  $t' \in subterms (last\ b) - base\text{-vars}\ b$ 
     $AT (t =_s t') \in set\ b \vee AT (t' =_s t) \in set\ b$ 
proof(cases  $t \in subterms (last\ b) - base\text{-vars}\ b$ )
case False
from assms have  $t \in subterms\ b$ 
  using base-vars-Un-subterms'-eq-subterms by auto
with False consider (t-base-vars)  $t \in base\text{-vars}\ b \mid (t\text{-wits})\ t \in Var\ ' wits\ b$ 
  using no-new-subterms-if-wf-branch[OF wf-branch]
  by (meson DiffI no-new-subterms-def)
then show ?thesis
proof cases

```

```

case t-wits
with  $\langle t \in \text{subterms}' b \rangle$  have  $t \notin \text{Var}$  ‘ pwits b
  unfolding subterms'-def base-vars-def by blast
with t-wits obtain t' where  $t': t' \in \text{subterms}$  (last b)
   $AT (t =_s t') \in \text{set } b \vee AT (t' =_s t) \in \text{set } b$ 
  unfolding pwits-def by blast
with  $\langle t \in \text{subterms}' b \rangle$  have  $t' \notin \text{base-vars } b$ 
  using AT-eq-urelems-subterms'-cases B-T-partition-verts(1)
  by (metis Un-iff base-vars-def disjoint-iff)
with t' that show ?thesis
  by blast
qed (use assms[unfolded subterms'-def] in blast)
qed

```

lemma *realisation-if-AF-eq*:

```

assumes sat b
assumes  $AF (t1 =_s t2) \in \text{set } b$ 
shows realise t1  $\neq$  realise t2

```

proof –

```

note AF-eq-branch-wits-subtermsD[OF assms(2)]

```

then consider

```

(t1-base-vars)  $t1 \in \text{base-vars } b$  |
(t2-base-vars)  $t2 \in \text{base-vars } b \vee t1 \in \text{base-vars } b \cup \text{subterms}' b$  |
(subterms)  $t1 \in \text{subterms}' b \vee t2 \in \text{subterms}' b$ 
by (metis UnE wf-branch wits-subterms-eq-base-vars-Un-subterms')

```

then show *?thesis*

proof *cases*

```

case t1-base-vars
with assms show ?thesis
  using realise-base-vars-if-AF-eq by blast

```

next

```

case t2-base-vars
with assms show ?thesis
  using realise-base-vars-if-AF-eq by blast

```

next

```

case subterms
define  $\Delta$  where  $\Delta \equiv \{(t1, t2) \in \text{subterms}' b \times \text{subterms}' b.$ 
   $AF (t1 =_s t2) \in \text{set } b \wedge \text{realise } t1 = \text{realise } t2\}$ 

```

have *finite* Δ

proof –

```

have  $\Delta \subseteq \text{subterms}' b \times \text{subterms}' b$ 
  unfolding  $\Delta$ -def by auto
moreover note finite-cartesian-product[OF finite-subterms' finite-subterms']
ultimately show ?thesis
  using finite-subset by blast

```

qed

```

let  $?h = \lambda(t1, t2). \text{min} (\text{height } t1) (\text{height } t2)$ 

```

```

have False if  $\Delta \neq \{\}$ 

```

```

proof –
  obtain  $t1\ t2$  where  $t1-t2: (t1, t2) = \text{arg-min-on } ?h\ \Delta$ 
    by (metis surj-pair)
  have  $(t1, t2) \in \Delta\ ?h\ (t1, t2) = \text{Min } (?h\ ' \Delta)$ 
  proof –
    from arg-min-if-finite(1)[OF <finite Δ> that] t1-t2 show  $(t1, t2) \in \Delta$ 
      by metis

    have  $f\ (\text{arg-min-on } f\ S) = \text{Min } (f\ ' S)$  if finite S S ≠ {}
      for  $f :: ('a\ \text{pset-term} \times 'a\ \text{pset-term}) \Rightarrow \text{nat}$  and  $S$ 
      using arg-min-least[OF that] that
      by (auto intro!: Min-eqI[symmetric] intro: arg-min-if-finite(1)[OF that])
    from this[OF <finite Δ> that] t1-t2 show  $?h\ (t1, t2) = \text{Min } (?h\ ' \Delta)$ 
      by auto
  qed
  then have  $*$ :  $t1 \in \text{subterms}'\ b\ t2 \in \text{subterms}'\ b$ 
     $AF\ (t1 =_s\ t2) \in \text{set } b$  realise t1 = realise t2
    unfolding  $\Delta$ -def by auto
  obtain  $t1'\ t2'$  where  $t1'-t2'$ :
     $t1' \in \text{subterms } (\text{last } b) - \text{base-vars } b\ t2' \in \text{subterms } (\text{last } b) - \text{base-vars } b$ 
     $AF\ (t1' =_s\ t2') \in \text{set } b$ 
    realise t1' = realise t1 realise t2' = realise t2
  proof –
    note Ex-AT-eq-mem-subterms-last-if-subterms'[OF <t1 ∈ subterms' b>]
    then obtain  $t1''$  where
       $t1'' \in \text{subterms } (\text{last } b) - \text{base-vars } b\ AF\ (t1'' =_s\ t2) \in \text{set } b$ 
      realise t1'' = realise t1
    proof cases
      case  $(2\ t1')$ 
      from bopen neqSelf <AF (t1 =_s t2) ∈ set b> have  $t1 \neq t2$ 
        by blast
      with  $2\ \langle \text{sat } b \rangle$  [THEN satD(1), THEN lin-satD] have  $AF\ (t1' =_s\ t2) \in \text{set } b$ 
        using lexpands-eq.intros(2,4)[OF - <AF (t1 =_s t2) ∈ set b>, THEN lexpands.intros(6)]
        by fastforce
      with that[OF - this] 2(1) <sat b>[unfolded sat-def] show ?thesis
        using realisation-if-AT-eq 2 by metis
    qed (use * that[of t1] in blast)
  moreover
    note Ex-AT-eq-mem-subterms-last-if-subterms'[OF <t2 ∈ subterms' b>]
    then obtain  $t2''$  where
       $t2'' \in \text{subterms } (\text{last } b) - \text{base-vars } b\ AF\ (t1'' =_s\ t2'') \in \text{set } b$ 
      realise t2'' = realise t2
    proof cases
      case  $(2\ t2')$ 
      from bopen neqSelf <AF (t1'' =_s t2) ∈ set b> have  $t1'' \neq t2$ 
        by blast
      with  $2\ \langle \text{sat } b \rangle$  [THEN satD(1), THEN lin-satD] have  $AF\ (t1'' =_s\ t2') \in \text{set } b$ 

```

```

set b
  using lexpands-eq.intros(2,4)[OF -  $\langle AF (t1'' =_s t2) \in set\ b \rangle$ , THEN
lexpands.intros(6)]
  by fastforce
  with that[OF - this] 2(1)  $\langle sat\ b \rangle$ [unfolded sat-def] show ?thesis
  using realisation-if-AT-eq 2 by metis
qed (use  $\langle AF (t1'' =_s t2) \in set\ b \rangle$  that[of t2] in blast)
ultimately show ?thesis
  using that * by metis
qed
with  $\langle realise\ t1 = realise\ t2 \rangle$  have  $(t1', t2') \in \Delta$ 
  unfolding  $\Delta$ -def subterms'-def
  by (simp add: AF-eq-subterms-branchD(1,2))
then have  $t1'-t2'$ -subterms:  $t1' \in subterms'\ b\ t2' \in subterms'\ b$ 
  unfolding  $\Delta$ -def by blast+

from  $t1'-t2'$  lemma1-2 *(3) have ?h  $(t1', t2') = ?h (t1, t2)$ 
by (metis in-subterms'-if-AF-eq-in-branch(1,2)[OF wf-branch] case-prod-conv)

from mem-urelems-if-urelem-last[OF wf-branch]  $t1'-t2'$ (1,2)
have not-urelem:  $\neg urelem (last\ b)\ t1' \neg urelem (last\ b)\ t2'$ 
  unfolding base-vars-def by auto
from finite-vars-branch infinite-vars obtain x where  $x \notin vars\ b$ 
  using ex-new-if-finite by blast
from bexpands-wit.intros[OF  $t1'-t2'$ (3) - - - this not-urelem]
   $t1'-t2'$ (1,2)  $\langle sat\ b \rangle$ [unfolded sat-def] consider
s1 where  $AT (s1 \in_s t1') \in set\ b\ AF (s1 \in_s t2') \in set\ b$  |
s2 where  $AF (s2 \in_s t1') \in set\ b\ AT (s2 \in_s t2') \in set\ b$ 
  using bexpands.intros(2-) by (metis Diff-iff)
then show ?thesis
proof cases
case 1
then have realise s1  $\in$  realise t2'
  using realisation-if-AT-mem
  by (metis *(4)  $t1'-t2'$ (4)  $t1'-t2'$ (5))
with 1  $t1'-t2'$ (3,4) obtain s2 where
   $s2 \rightarrow_{bgraph\ b}\ t2'\ realise\ s1 = realise\ s2$ 
using dominates-if-mem-realisation-in-subterms'-if-AT-mem-in-branch(1)[OF
wf-branch]
  by metis
then have  $AT (s2 \in_s t2') \in set\ b$ 
  unfolding bgraph-def Let-def by auto
with  $\langle AF (s1 \in_s t2') \in set\ b \rangle$   $\langle sat\ b \rangle$ [THEN satD(1), THEN lin-satD]
have  $AF (s2 =_s s1) \in set\ b$ 
  using lexpands-eq.intros(5)[THEN lexpands.intros(6)] by fastforce
then have  $s1 \neq s2$ 
  using neqSelf bopen by blast
from realise-base-vars-if-AF-eq[OF  $\langle sat\ b \rangle$   $\langle AF (s2 =_s s1) \in set\ b \rangle$ 
 $\langle realise\ s1 = realise\ s2 \rangle$ 

```

have $s1 \in \text{subterms}' b$ $s2 \in \text{subterms}' b$
by (metis Un-iff $\langle AF (s2 =_s s1) \in \text{set } b \rangle$ in-subterms'-if-AF-eq-in-branch
wf-branch)+

with $\langle \text{realise } s1 = \text{realise } s2 \rangle$ **have** $(s2, s1) \in \Delta$
unfolding Δ -def **using** $\langle AF (s2 =_s s1) \in \text{set } b \rangle$ **by** auto
moreover
have $\text{realise } s1 \in \text{realise } t1'$ $\text{realise } s2 \in \text{realise } t1'$
 $\text{realise } s1 \in \text{realise } t2'$ $\text{realise } s2 \in \text{realise } t2'$
using $\langle \text{realise } s1 \in \text{realise } t2' \rangle$ $\langle \text{realise } s1 = \text{realise } s2 \rangle$
using $*(4)$ $t1'-t2'(4,5)$ **by** auto
with lemma1-3 **have** $?h (s2, s1) < ?h (t1', t2')$
using $\langle s1 \in \text{subterms}' b \rangle$ $\langle s2 \in \text{subterms}' b \rangle$ $t1'-t2'$ -subterms
by (auto simp: min-def)
ultimately show ?thesis
using arg-min-least[OF $\langle \text{finite } \Delta \rangle$ $\langle \Delta \neq \{\} \rangle$]
using $\langle (t1', t2') \in \Delta \rangle$ $\langle ?h (t1', t2') = ?h (t1, t2) \rangle$ $t1-t2$
by (metis (mono-tags, lifting) le-antisym le-eq-less-or-eq nat-neq-iff)

next

case 2
then have $\text{realise } s2 \in \text{realise } t1'$
using realisation-if-AT-mem
by (metis $*(4)$ $t1'-t2'(4)$ $t1'-t2'(5)$)
with 2 $t1'-t2'(3,4)$ **obtain** $s1$ **where**
 $s1 \rightarrow_{\text{bgraph } b} t1'$ $\text{realise } s1 = \text{realise } s2$
using dominates-if-mem-realisation **by** metis
then have $AT (s1 \in_s t1') \in \text{set } b$
unfolding bgraph-def Let-def **by** auto
with $\langle AF (s2 \in_s t1') \in \text{set } b \rangle$ $\langle \text{sat } b \rangle$ [unfolded sat-def]
have $AF (s1 =_s s2) \in \text{set } b$
using lexpands-eq.intros(5)[THEN lexpands.intros(6)]
using lin-satD **by** fastforce
then have $s1 \neq s2$
using neqSelf bopen **by** blast
from realise-base-vars-if-AF-eq[OF $\langle \text{sat } b \rangle$ $\langle AF (s1 =_s s2) \in \text{set } b \rangle$]
 $\langle \text{realise } s1 = \text{realise } s2 \rangle$
have $s1 \in \text{subterms}' b$ $s2 \in \text{subterms}' b$
by (metis Un-iff $\langle AF (s1 =_s s2) \in \text{set } b \rangle$ in-subterms'-if-AF-eq-in-branch
wf-branch)+

with $\langle \text{realise } s1 = \text{realise } s2 \rangle$ **have** $(s1, s2) \in \Delta$
unfolding Δ -def **using** $\langle AF (s1 =_s s2) \in \text{set } b \rangle$ **by** auto
moreover
have $\text{realise } s1 \in \text{realise } t1'$ $\text{realise } s2 \in \text{realise } t1'$
 $\text{realise } s1 \in \text{realise } t2'$ $\text{realise } s2 \in \text{realise } t2'$
using $\langle \text{realise } s2 \in \text{realise } t1' \rangle$ $\langle \text{realise } s1 = \text{realise } s2 \rangle$
using $*(4)$ $t1'-t2'(4,5)$ **by** auto
with lemma1-3 **have** $?h (s1, s2) < ?h (t1', t2')$
using $\langle s1 \in \text{subterms}' b \rangle$ $\langle s2 \in \text{subterms}' b \rangle$ $t1'-t2'$ -subterms

```

    by (auto simp: min-def)
  ultimately show ?thesis
    using arg-min-least[OF ‹finite  $\Delta$ › ‹ $\Delta \neq \{\}$ ›]
    using ‹ $(t1', t2') \in \Delta$ › ‹ $?h (t1', t2') = ?h (t1, t2)$ › t1-t2
    by (metis (mono-tags, lifting) le-antisym le-eq-less-or-eq nat-neq-iff)
  qed
  qed
  with assms subterms show ?thesis
    unfolding  $\Delta$ -def by blast
  qed
  qed

lemma realisation-if-AF-mem:
  assumes sat b
  assumes AF  $(s \in_s t) \in \text{set } b$ 
  shows realise s  $\notin$  realise t
proof
  assume realise s  $\in$  realise t
  from assms have s  $\in$  base-vars b  $\cup$  subterms' b
    t  $\in$  base-vars b  $\cup$  subterms' b
    using in-subterms'-if-AF-mem-in-branch[OF wf-branch] by blast+
  from dominates-if-mem-realisation[OF ‹realise s  $\in$  realise t›]
  obtain s' where s'  $\rightarrow_{\text{bgraph } b}$  t realise s = realise s'
    by blast
  then have AT  $(s' \in_s t) \in \text{set } b$ 
    unfolding bgraph-def Let-def by auto
  with assms lexpands-eq.intros(5)[THEN lexpands.intros(6)] have AF  $(s' =_s s)$ 
 $\in \text{set } b$ 
    unfolding sat-def using lin-satD by fastforce
  from realisation-if-AF-eq[OF ‹sat b› this] ‹realise s = realise s'› show False
    by simp
  qed

lemma realisation-Empty: realise  $(\emptyset n) = 0$ 
proof -
  from bopen have AT  $(s \in_s \emptyset n) \notin \text{set } b$  for s
    using bclosed.intros by blast
  then have parents (bgraph b)  $(\emptyset n) = \{\}$ 
    unfolding bgraph-def Let-def by auto
  moreover
  have  $(\emptyset n) \notin \text{base-vars } b$ 
  proof -
    have  $(\emptyset n) \notin \text{Var } \text{'pwits } b$ 
      using pwits-def wits-def by blast
    moreover have  $(\emptyset n) \notin \text{urelems } b$ 
      unfolding urelems-def using wf-branch[THEN wf-branch-not-Nil] last-in-set
      using is-Var-if-urelem' by fastforce
    ultimately show ?thesis
      unfolding base-vars-def by blast
  qed

```

qed
then have $(\emptyset n) \in \text{subterms}' b \vee (\emptyset n) \notin \text{verts} (\text{bgraph } b)$
by *(simp add: verts-bgraph)*
ultimately show $\text{realise } (\emptyset n) = 0$
by *(auto simp: verts-bgraph)*
qed

lemma *realisation-Union:*

assumes *sat b*
assumes $t1 \sqcup_s t2 \in \text{subterms } b$
shows $\text{realise } (t1 \sqcup_s t2) = \text{realise } t1 \sqcup \text{realise } t2$
using *assms*
proof –
from *assms* **have** *mem-subterms-last: t1 \sqcup_s t2 \in subterms (last b)*
using *mem-subterms-fm-last-if-mem-subterms-branch[OF wf-branch]*
by *simp*
note *not-urelem-comps-if-compound[where ?f=(\sqcup_s), OF assms(2), simplified]*
moreover note *subterms-fmD(1,2)[OF mem-subterms-last]*
then have $t1 \notin \text{Var } \text{'pwits } b$ $t2 \notin \text{Var } \text{'pwits } b$
unfolding *pwits-def wits-def*
using *pset-term.set-intros(1) mem-vars-fm-if-mem-subterms-fm* **by** *fastforce+*
ultimately have $t1 \in \text{subterms}' b$ $t2 \in \text{subterms}' b$
unfolding *subterms'-def base-vars-def*
using *assms(2)* **by** *(auto dest: subterms-branchD)*

from *assms(2)* **have** $t1 \sqcup_s t2 \in \text{subterms}' b$
using *base-vars-subs-vars base-vars-Un-subterms'-eq-subterms* **by** *blast*

have $\text{realise } (t1 \sqcup_s t2) \leq \text{realise } t1 \sqcup \text{realise } t2$

proof

fix *e* **assume** $e \in \text{realise } (t1 \sqcup_s t2)$
then obtain *s* **where** $s: e = \text{realise } s$ $s \rightarrow_{\text{bgraph } b} (t1 \sqcup_s t2)$
using *dominates-if-mem-realisation <t1 \sqcup_s t2 \in subterms' b>*
by *auto*
then have $AT (s \in_s t1 \sqcup_s t2) \in \text{set } b$
unfolding *bgraph-def Let-def* **by** *auto*
with $\langle \text{sat } b \rangle$ **consider** $AT (s \in_s t1) \in \text{set } b \mid AF (s \in_s t1) \in \text{set } b$
unfolding *sat-def* **using** *beexpands-nowit.intros(3)[OF - mem-subterms-last,*
THEN beexpands.intros(1)]
by *blast*
then show $e \in \text{realise } t1 \sqcup \text{realise } t2$
proof(*cases*)
case *1*
with *s* **show** *?thesis* **using** *realisation-if-AT-mem* **by** *auto*
next
case *2*
from $\langle \text{sat } b \rangle$ *lexpands-un.intros(4)[OF <AT (s \in_s t1 \sqcup_s t2) \in set b> this]*
have $AT (s \in_s t2) \in \text{set } b$
unfolding *sat-def* **using** *lin-satD lexpands.intros(2)* **by** *force*

with s show *?thesis* using *realisation-if-AT-mem* by *auto*
 qed
 qed
 moreover have $\text{realise } t1 \sqcup \text{realise } t2 \leq \text{realise } (t1 \sqcup_s t2)$
 proof
 fix e assume $e \in \text{realise } t1 \sqcup \text{realise } t2$
 with $\langle t1 \in \text{subterms}' b \rangle \langle t2 \in \text{subterms}' b \rangle$ consider
 $s1$ where $e = \text{realise } s1 \ s1 \rightarrow_{\text{bgraph } b} t1$ |
 $s2$ where $e = \text{realise } s2 \ s2 \rightarrow_{\text{bgraph } b} t2$
 using *dominates-if-mem-realisation* by *force*
 then show $e \in \text{realise } (t1 \sqcup_s t2)$
 proof(*cases*)
 case 1
 then have $AT (s1 \in_s t1) \in \text{set } b$
 unfolding *bgraph-def Let-def* by *auto*
 from $\langle \text{sat } b \rangle$ *lexpands-un.intros(2)*[*OF this mem-subterms-last, THEN lex-*
pands.intros(2)]
 have $AT (s1 \in_s t1 \sqcup_s t2) \in \text{set } b$
 unfolding *sat-def* using *lin-satD* by *force*
 with 1 *realisation-if-AT-mem*[*OF this*] show *?thesis*
 by *blast*
 next
 case 2
 then have $AT (s2 \in_s t2) \in \text{set } b$
 unfolding *bgraph-def Let-def* by *auto*
 from $\langle \text{sat } b \rangle$ *lexpands-un.intros(3)*[*OF this mem-subterms-last, THEN lex-*
pands.intros(2)]
 have $AT (s2 \in_s t1 \sqcup_s t2) \in \text{set } b$
 unfolding *sat-def* using *lin-satD* by *force*
 with 2 *realisation-if-AT-mem*[*OF this*] show *?thesis*
 by *blast*
 qed
 qed
 ultimately show *?thesis* by *blast*
 qed

lemma *realisation-Inter*:

assumes *sat b*
 assumes $t1 \sqcap_s t2 \in \text{subterms } b$
 shows $\text{realise } (t1 \sqcap_s t2) = \text{realise } t1 \sqcap \text{realise } t2$
 using *assms*
 proof –
 from *assms* have *mem-subterms-last*: $t1 \sqcap_s t2 \in \text{subterms } (last\ b)$
 using *mem-subterms-fm-last-if-mem-subterms-branch*[*OF wf-branch*]
 by *simp*
 note *not-urelem-comps-if-compound*[**where** $?f=(\sqcap_s)$, *OF assms(2)*, *simplified*]
 moreover note *subterms-fmD(3,4)*[*OF mem-subterms-last*]
 then have $t1 \notin \text{Var } 'pwits\ b \ t2 \notin \text{Var } 'pwits\ b$
 unfolding *pwits-def wits-def*

using *pset-term.set-intros(1) mem-vars-fm-if-mem-subterms-fm* **by** *fastforce+*
ultimately have *t1-t2-subterms'*: $t1 \in \text{subterms}' b$ $t2 \in \text{subterms}' b$
unfolding *subterms'-def base-vars-def*
using *assms(2)* **by** (*auto dest: subterms-branchD*)

from *assms(2)* **have** $t1 \sqcap_s t2 \in \text{subterms}' b$
using *base-vars-subs-vars base-vars-Un-subterms'-eq-subterms* **by** *blast*

have *realise* $(t1 \sqcap_s t2) \leq \text{realise } t1 \sqcap \text{realise } t2$
proof
fix *e* **assume** $e \in \text{realise } (t1 \sqcap_s t2)$
with $\langle t1 \sqcap_s t2 \in \text{subterms}' b \rangle$ **obtain** *s*
where $s: e = \text{realise } s \ s \rightarrow_{\text{bgraph } b} (t1 \sqcap_s t2)$
using *dominates-if-mem-realisation* **by** *auto*
then have $AT (s \in_s t1 \sqcap_s t2) \in \text{set } b$
unfolding *bgraph-def Let-def* **by** *auto*
with $\langle \text{sat } b \rangle$ *lexpands-int.intros(1)* [*OF this, THEN lexpands.intros(3)*]
have $AT (s \in_s t1) \in \text{set } b$ $AT (s \in_s t2) \in \text{set } b$
unfolding *sat-def* **using** *lin-satD* **by** *force+*
from *this* [*THEN realisation-if-AT-mem*] *s* **show** $e \in \text{realise } t1 \sqcap \text{realise } t2$
by *auto*

qed
moreover have *realise* $t1 \sqcap \text{realise } t2 \leq \text{realise } (t1 \sqcap_s t2)$
proof
fix *e* **assume** $e \in \text{realise } t1 \sqcap \text{realise } t2$
with $\langle t1 \in \text{subterms}' b \rangle$ $\langle t2 \in \text{subterms}' b \rangle$ **obtain** *s1 s2* **where** *s1-s2*:
 $e = \text{realise } s1 \ s1 \rightarrow_{\text{bgraph } b} t1$
 $e = \text{realise } s2 \ s2 \rightarrow_{\text{bgraph } b} t2$
using *dominates-if-mem-realisation* **by** *auto metis*
then have $AT (s1 \in_s t1) \in \text{set } b$ $AT (s2 \in_s t2) \in \text{set } b$
unfolding *bgraph-def Let-def* **by** *auto*
moreover have $AT (s1 \in_s t2) \in \text{set } b$
proof –
from $\langle \text{sat } b \rangle$ **have** $AT (s1 \in_s t2) \in \text{set } b \vee AF (s1 \in_s t2) \in \text{set } b$
unfolding *sat-def*
using *beexpands-nowit.intros(4)* [*OF* $\langle AT (s1 \in_s t1) \in \text{set } b \rangle$ *mem-subterms-last*]
using *beexpands.intros(1)* **by** *blast*
moreover from $\langle \text{sat } b \rangle$ *s1-s2* **have** *False* **if** $AF (s1 \in_s t2) \in \text{set } b$
proof –
note *lexpands-eq.intros(5)* [*OF* $\langle AT (s2 \in_s t2) \in \text{set } b \rangle$ *that, THEN lexpands.intros(6)*]
with *realisation-if-AF-eq* [*OF* $\langle \text{sat } b \rangle$, *of* *s2 s1*] **have** *realise* $s2 \neq \text{realise } s1$
using $\langle \text{sat } b \rangle$ **by** (*auto simp: sat-def lin-satD*)
with $\langle e = \text{realise } s1 \rangle$ $\langle e = \text{realise } s2 \rangle$ **show** *False* **by** *simp*

qed
ultimately show $AT (s1 \in_s t2) \in \text{set } b$ **by** *blast*

qed
ultimately have $AT (s1 \in_s t1 \sqcap_s t2) \in \text{set } b$
using $\langle \text{sat } b \rangle$ *lexpands-int.intros(6)* [*OF* - - *mem-subterms-last, THEN le-*

pands.intros(3)
unfolding *sat-def* **by** (*fastforce simp: lin-satD*)
from *realisation-if-AT-mem*[*OF this*] **show** $e \in \text{realise } (t1 \sqcap_s t2)$
unfolding $\langle e = \text{realise } s1 \rangle$
by *simp*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *realisation-Diff*:
assumes *sat b*
assumes $s -_s t \in \text{subterms } b$
shows $\text{realise } (s -_s t) = \text{realise } s - \text{realise } t$
using *assms*
proof –
from *assms* **have** *mem-subterms-last*: $s -_s t \in \text{subterms } (\text{last } b)$
using *mem-subterms-fm-last-if-mem-subterms-branch*[*OF wf-branch*]
by *simp*
note *not-urelem-comps-if-compound*[**where** $?f = (-_s)$, *OF assms(2)*, *simplified*]
moreover note *subterms-fmD(5,6)*[*OF mem-subterms-last*]
then have $s \notin \text{Var } \langle \text{pwits } b \ t \notin \text{Var } \langle \text{pwits } b \rangle \rangle$
unfolding *pwits-def wits-def*
using *pset-term.set-intros(1)* *mem-vars-fm-if-mem-subterms-fm* **by** *fastforce+*
ultimately have $s \in \text{subterms}' b \ t \in \text{subterms}' b$
unfolding *subterms'-def base-vars-def*
using *assms(2)* **by** (*auto dest: subterms-branchD*)

from *assms(2)* **have** $s -_s t \in \text{subterms}' b$
using *base-vars-subs-vars base-vars-Un-subterms'-eq-subterms* **by** *blast*

have $\text{realise } (s -_s t) \leq \text{realise } s - \text{realise } t$
proof
fix *e* **assume** $e \in \text{realise } (s -_s t)$
then obtain *u* **where** $u: e = \text{realise } u \ u \rightarrow_{\text{bgraph } b} (s -_s t)$
using *dominates-if-mem-realisation* $\langle s -_s t \in \text{subterms}' b \rangle$ **by** *auto*
then have $AT (u \in_s s -_s t) \in \text{set } b$
unfolding *bgraph-def Let-def* **by** *auto*
with $\langle \text{sat } b \rangle$ *lexpands-diff.intros(1)*[*OF this*, *THEN lexpands.intros(4)*]
have $AT (u \in_s s) \in \text{set } b \ AF (u \in_s t) \in \text{set } b$
unfolding *sat-def* **using** *lin-satD* **by** *force+*
with *u* **show** $e \in \text{realise } s - \text{realise } t$
using $\langle \text{sat } b \rangle$ *realisation-if-AT-mem* *realisation-if-AF-mem*
by *auto*
qed
moreover have $\text{realise } s - \text{realise } t \leq \text{realise } (s -_s t)$
proof
fix *e* **assume** $e \in \text{realise } s - \text{realise } t$
then obtain *u* **where** $u:$
 $e = \text{realise } u \ u \rightarrow_{\text{bgraph } b} s \ \neg \ u \rightarrow_{\text{bgraph } b} t$

using *dominates-if-mem-realisation* $\langle s \in \text{subterms}' b \rangle \langle t \in \text{subterms}' b \rangle$ **by**
auto
then have $AT (u \in_s s) \in \text{set } b$
unfolding *bgraph-def Let-def* **by** *auto*
moreover have $AF (u \in_s t) \in \text{set } b$
proof –
from $\langle \text{sat } b \rangle$ **have** $AT (u \in_s t) \in \text{set } b \vee AF (u \in_s t) \in \text{set } b$
unfolding *sat-def* **using** *bexpands-nowit.intros(5)* [*OF* $\langle AT (u \in_s s) \in \text{set } b \rangle$ *mem-subterms-last*]
using *bexpands.intros(1)* **by** *blast*
moreover from $u(3)$ **have** *False* **if** $AT (u \in_s t) \in \text{set } b$
using *that unfolding Let-def bgraph-def* **by** (*auto simp: arcs-ends-def arc-to-ends-def*)
ultimately show $AF (u \in_s t) \in \text{set } b$
by *blast*
qed
ultimately have $AT (u \in_s s -_s t) \in \text{set } b$
using $\langle \text{sat } b \rangle$ *lexpands-diff.intros(6)* [*OF* - - *mem-subterms-last, THEN lexpands.intros(4)*]
unfolding *sat-def* **by** (*fastforce simp: lin-satD*)
from *realisation-if-AT-mem* [*OF this*] **show** $e \in \text{realise } (s -_s t)$
unfolding $\langle e = \text{realise } u \rangle$
by *simp*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *realisation-Single*:

assumes *sat b*
assumes *Single t ∈ subterms b*
shows $\text{realise } (\text{Single } t) = HF \{ \text{realise } t \}$
using *assms*
proof –
from *assms* **have** *mem-subterms-last: Single t ∈ subterms (last b)*
using *mem-subterms-fm-last-if-mem-subterms-branch* [*OF wf-branch*]
by *simp*
have *Single t ∈ subterms' b*
proof –
from *urelems-subs-vars* **have** *Single t ∉ base-vars b*
unfolding *base-vars-def* **by** *blast*
then show *?thesis*
by (*simp add: assms(2) subterms'-def*)
qed

have $\text{realise } (\text{Single } t) \leq HF \{ \text{realise } t \}$

proof

fix e **assume** $e \in \text{realise } (\text{Single } t)$

then obtain s **where** $s: e = \text{realise } s \ s \rightarrow_{\text{bgraph } b} \text{Single } t$

using *dominates-if-mem-realisation* $\langle \text{Single } t \in \text{subterms}' b \rangle$ **by** *auto*

```

then have AT (s ∈s Single t) ∈ set b
  unfolding bgraph-def Let-def by auto
with ⟨sat b⟩ lexpands-single.intros(2)[OF this, THEN lexpands.intros(5)]
have AT (s =s t) ∈ set b
  unfolding sat-def using lin-satD by fastforce
with s show e ∈ HF {realise t}
  using realisation-if-AT-eq ⟨sat b⟩[unfolded sat-def]
  by auto
qed
moreover have HF {realise t} ≤ realise (Single t)
proof
  fix e assume e ∈ HF {realise t}
  then have e = realise t
    by simp
  from lexpands-single.intros(1)[OF mem-subterms-last, THEN lexpands.intros(5)]
  ⟨sat b⟩
  have AT (t ∈s Single t) ∈ set b
    unfolding sat-def using lin-satD by fastforce
  from realisation-if-AT-mem[OF this] ⟨e = realise t⟩
  show e ∈ realise (Single t)
    by simp
qed
ultimately show ?thesis by blast
qed

lemmas realisation-simps =
  realisation-Empty realisation-Union realisation-Inter realisation-Diff realisation-Single

end

```

Coherence

```

lemma (in open-branch) Ist-realisation-eq-realisation:
  assumes sat b t ∈ subterms b
  shows Ist (λx. realise (Var x)) t = realise t
  using assms
  by (induction t) (force simp: realisation-simps dest: subterms-branchD)+

lemma (in open-branch) coherence:
  assumes sat b φ ∈ set b
  shows interp Isa (λx. realise (Var x)) φ
  using assms
proof(induction size φ arbitrary: φ rule: less-induct)
  case less
  then show ?case
  proof(cases φ)
    case (Atom a)
    then show ?thesis
    proof(cases a)

```

```

    case (Elem s t)
    with Atom less have s ∈ subterms b t ∈ subterms b
      using AT-mem-subterms-branchD by blast+
    with Atom Elem less show ?thesis
      using Ist-realisation-eq-realisation[OF ‹sat b›] realisation-if-AT-mem by
auto
  next
  case (Equal s t)
  with Atom less have s ∈ subterms b t ∈ subterms b
    using AT-eq-subterms-branchD by blast+
  with Atom Equal less satD(1)[OF ‹sat b›] show ?thesis
    using Ist-realisation-eq-realisation[OF ‹sat b›] realisation-if-AT-eq by auto
  qed
next
case (And φ1 φ2)
with ‹φ ∈ set b› ‹sat b›[THEN satD(1), THEN lin-satD] have φ1 ∈ set b φ2
∈ set b
  using lexpands-fm.intros(1)[THEN lexpands.intros(1)] by fastforce+
with And less show ?thesis by simp
next
case (Or φ1 φ2)
with ‹φ ∈ set b› ‹sat b›[THEN satD(2)] have φ1 ∈ set b ∨ Neg φ1 ∈ set b
  using bexpands-nowit.intros(1)[THEN bexpands.intros(1)]
  by blast
with less Or ‹sat b›[THEN satD(1), THEN lin-satD] have φ1 ∈ set b ∨ φ2 ∈
set b
  using lexpands-fm.intros(3)[THEN lexpands.intros(1)] by fastforce
with Or less show ?thesis
  by force
next
case (Neg φ′)
show ?thesis
proof(cases φ′)
  case (Atom a)
  then show ?thesis
  proof(cases a)
    case (Elem s t)
    with Atom Neg less have s ∈ subterms b t ∈ subterms b
      using AF-mem-subterms-branchD by blast+
    with Neg Atom Elem less show ?thesis
      using Ist-realisation-eq-realisation realisation-if-AF-mem ‹sat b› by auto
  next
  case (Equal s t)
  with Atom Neg less have s ∈ subterms b t ∈ subterms b
    using AF-eq-subterms-branchD by blast+
  with Neg Atom Equal less show ?thesis
    using Ist-realisation-eq-realisation realisation-if-AF-eq ‹sat b› by auto
  qed
next

```

```

case (And  $\varphi1$   $\varphi2$ )
with Neg  $\langle \text{sat } b \rangle$  [THEN satD(2)] less have  $\varphi1 \in \text{set } b \vee \text{Neg } \varphi1 \in \text{set } b$ 
  using bexpands-nowit.intros(2) [THEN bexpands.intros(1)] by blast
with  $\langle \text{sat } b \rangle$  [THEN satD(1), THEN lin-satD] Neg And less
have Neg  $\varphi2 \in \text{set } b \vee \text{Neg } \varphi1 \in \text{set } b$ 
  using lexpands-fm.intros(5) [THEN lexpands.intros(1)] by fastforce
with Neg And less show ?thesis by force
next
case (Or  $\varphi1$   $\varphi2$ )
with  $\langle \varphi \in \text{set } b \rangle$  Neg  $\langle \text{sat } b \rangle$  [THEN satD(1), THEN lin-satD]
have Neg  $\varphi1 \in \text{set } b$  Neg  $\varphi2 \in \text{set } b$ 
  using lexpands-fm.intros(2) [THEN lexpands.intros(1)] by fastforce+
moreover have size (Neg  $\varphi1$ ) < size  $\varphi$  size (Neg  $\varphi2$ ) < size  $\varphi$ 
  using Neg Or by simp-all
ultimately show ?thesis using Neg Or less by force
next
case Neg': (Neg  $\varphi''$ )
with  $\langle \varphi \in \text{set } b \rangle$  Neg  $\langle \text{sat } b \rangle$  [THEN satD(1), THEN lin-satD] have  $\varphi'' \in \text{set } b$ 
  using lexpands-fm.intros(7) [THEN lexpands.intros(1)] by fastforce+
with Neg Neg' less show ?thesis by simp
qed
qed
qed

```

4.2.3 Soundness of the Calculus

Soundness of Closedness

lemmas *wf-trancl-hmem-rel = wf-trancl[OF wf-hmem-rel]*

lemma *trancl-hmem-rel-not-refl*: $(x, x) \notin \text{hmem-rel}^+$
using *wf-trancl-hmem-rel* **by** *simp*

lemma *mem-trancl-elts-rel-if-member-seq*:

assumes *member-seq* s cs t

assumes $cs \neq []$

assumes $\forall a \in \text{set } cs. I_{sa} M a$

shows $(I_{st} M s, I_{st} M t) \in \text{hmem-rel}^+$

using *assms*

proof(*induction rule: member-seq.induct*)

case (2 s s' u cs t)

show *?case*

proof(*cases* cs)

case *Nil*

with 2 **show** *?thesis*

unfolding *hmem-rel-def* **by** *auto*

next

case (*Cons* c cs')

with 2 **have** $(I_{st} M u, I_{st} M t) \in \text{hmem-rel}^+$

by *simp*
 moreover from \mathcal{Q} have $I_{sa} M (s \in_s u)$
 by *simp*
 ultimately show *?thesis*
 unfolding *hmem-rel-def* by (*simp add: trancl-into-trancl2*)
 qed
 qed *simp-all*

lemma *bclosed-sound*:
 assumes *bclosed b*
 shows $\exists \varphi \in \text{set } b. \neg \text{interp } I_{sa} M \varphi$
 using *assms*
proof –
 have *False* if $\forall \varphi \in \text{set } b. \text{interp } I_{sa} M \varphi$
 using *assms that*
proof(*induction rule: bclosed.induct*)
 case (*memberCycle cs b*)
 then have $\forall a \in \text{set } cs. I_{sa} M a$
 unfolding *Atoms-def* by *fastforce*
 from *memberCycle* obtain *s* where *member-seq s cs s*
 using *member-cycle.elims(2)* by *blast*
 with *memberCycle* $\langle \forall a \in \text{set } cs. I_{sa} M a \rangle$ have $(I_{st} M s, I_{st} M s) \in \text{hmem-rel}^+$
 using *mem-trancl-elts-rel-if-member-seq member-cycle.simps(2)* by *blast*
 with *trancl-hmem-rel-not-refl* show *?case*
 by *blast*
 qed (*use interp.simps(4) in <fastforce+>*)
 then show *?thesis*
 by *blast*
 qed

lemma *types-term-lt-if-member-seq*:
 includes *no member-ASCII-syntax*
 fixes *cs* :: 'a *pset-atom list*
 assumes $\forall a \in \text{set } cs. v \vdash a$
 assumes *member-seq s cs t cs* $\neq []$
 shows $\exists ls \ lt. v \vdash s : ls \wedge v \vdash t : lt \wedge ls < lt$
 using *assms*
proof(*induction s cs t rule: member-seq.induct*)
 case ($2\ s\ s'\ u\ cs\ t$)
 then show *?case*
proof(*cases cs*)
 case (*Cons c cs'*)
 with \mathcal{Q} obtain *lu lt* where $v \vdash u : lu\ v \vdash t : lt\ lu < lt$
 by *auto*
 moreover from \mathcal{Q} obtain *ls* where $v \vdash s : ls\ ls < lu$
 using $\langle v \vdash u : lu \rangle$ by (*auto simp: types-pset-atom.simps dest: types-term-unique*)
 ultimately show *?thesis*
 by *fastforce*
 qed (*fastforce simp: types-pset-atom.simps*)

qed *auto*

lemma *no-member-cycle-if-types-last*:

fixes $b :: 'a \text{ branch}$

assumes *wf-branch* b

assumes $\exists v. v \vdash \text{last } b$

shows $\neg (\text{member-cycle } cs \wedge \text{set } cs \subseteq \text{Atoms } (\text{set } b))$

proof

presume *member-cycle* $cs \text{ set } cs \subseteq \text{Atoms } (\text{set } b)$

then obtain s **where** *member-seq* $s \text{ cs } s \text{ cs} \neq []$

using *member-cycle.elims*(2) **by** *blast*

moreover from *assms* **obtain** v **where** $\forall \varphi \in \text{set } b. v \vdash \varphi$

using *types-urelems* **by** *blast*

with $\langle \text{set } cs \subseteq \text{Atoms } (\text{set } b) \rangle$ **have** $\forall a \in \text{set } cs. v \vdash a$

unfolding *Atoms-def* **by** (*auto dest!*: *types-fmD*(6))

ultimately show *False*

using *types-term-lt-if-member-seq types-term-unique* **by** *blast*

qed *simp-all*

Soundness of the Expansion Rules

lemma *lexpands-sound*:

assumes *lexpands* $b' b$

assumes $\varphi \in \text{set } b'$

assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{sa} M \psi$

shows *interp* $I_{sa} M \varphi$

using *assms*

proof(*induction rule: lexpands.induct*)

case (1 $b' b$)

then show *?case*

by (*induction rule: lexpands-fm.induct*)

(*metis empty-iff empty-set interp.simps*(2,3,4) *set-ConsD*)**+**

next

case (2 $b' b$)

then show *?case*

proof(*induction rule: lexpands-un.induct*)

case (4 $s \ t1 \ t2 \ \text{branch}$)

with *this*(1)[*THEN this*(4)] **show** *?case*

by *force*

next

case (5 $s \ t1 \ t2 \ \text{branch}$)

with *this*(1)[*THEN this*(4)] **show** *?case*

by *force*

qed *force+*

next

case (3 $b' b$)

then show *?case*

proof(*induction rule: lexpands-int.induct*)

case (4 $s \ t1 \ t2 \ \text{branch}$)

```

    with this(1)[THEN this(4)] show ?case
      by force
  next
    case (5 s t1 t2 branch)
    with this(1)[THEN this(4)] show ?case
      by force
  qed force+
next
  case (4 b' b)
  then show ?case
  proof(induction rule: lexpands-diff.induct)
    case (4 s t1 t2 branch)
    with this(1)[THEN this(4)] show ?case by force
  next
    case (5 s t1 t2 branch)
    with this(1)[THEN this(4)] show ?case by force
  qed force+
next
  case (5 b' b)
  then show ?case
  by (induction rule: lexpands-single.induct) force+
next
  case (6 b' b)
  then show ?case
  proof(induction rule: lexpands-eq.induct')
    case (subst t1 t2 t1' t2' p l b)
    with this(1,2)[THEN this(6)] show ?case
      by (cases l; cases p) auto
  next
    case (neq s t s' b)
    with this(1,2)[THEN this(4)] show ?case by force
  qed
qed

```

lemma *bexpands-nowit-sound*:
assumes *bexpands-nowit bs' b*
assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{s_a} M \psi$
shows $\exists b' \in bs'. \forall \psi \in \text{set } b'. \text{interp } I_{s_a} M \psi$
using *assms*
by (*induction rule: bexpands-nowit.induct*) *force+*

lemma *I_{st}-upd-eq-if-not-mem-vars-term*:
assumes $x \notin \text{vars } t$
shows $I_{st} (M(x := y)) t = I_{st} M t$
using *assms* **by** (*induction t*) *auto*

lemma *I_{sa}-upd-eq-if-not-mem-vars-atom*:
assumes $x \notin \text{vars } a$
shows $I_{sa} (M(x := y)) a = I_{sa} M a$

using *assms*
by (*cases a*) (*auto simp: I_{st}-upd-eq-if-not-mem-vars-term*)

lemma *interp-upd-eq-if-not-mem-vars-fm*:
assumes $x \notin \text{vars } \varphi$
shows $\text{interp } I_{sa} (M(x := y)) \varphi = \text{interp } I_{sa} M \varphi$
using *assms*
by (*induction* φ) (*auto simp: I_{sa}-upd-eq-if-not-mem-vars-atom*)

lemma *bexpands-wit-sound*:
assumes *bexpands-wit s t x bs' b*
assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{sa} M \psi$
shows $\exists M. \exists b' \in bs'. \forall \psi \in \text{set } (b' @ b). \text{interp } I_{sa} M \psi$
using *assms*

proof (*induction rule: bexpands-wit.induct*)
let $?bs' = \{[AT (Var x \in_s s), AF (Var x \in_s t)],$
 $[AT (Var x \in_s t), AF (Var x \in_s s)]\}$
case (1 *b*)
with *this(1)[THEN this(9)]* **have** $I_{st} M s \neq I_{st} M t$
by *auto*
then obtain *y* **where** *y*:
 $y \in I_{st} M s \wedge y \notin I_{st} M t \vee$
 $y \in I_{st} M t \wedge y \notin I_{st} M s$
by *auto*
have $x \notin \text{vars } s \wedge x \notin \text{vars } t$
using 1 **by** (*auto simp: vars-fm-vars-branchI*)
then have $I_{st} (M(x := y)) s = I_{st} M s \wedge I_{st} (M(x := y)) t = I_{st} M t$
using *I_{st}-upd-eq-if-not-mem-vars-term* **by** *metis+*
then have $\exists b' \in ?bs'. \forall \psi \in \text{set } b'. \text{interp } I_{sa} (M(x := y)) \psi$
using 1 *y* **by** *auto*
moreover have $\forall \psi \in \text{set } b. \text{interp } I_{sa} (M(x := y)) \psi$
using 1(9) $\langle x \notin \text{vars } b \rangle$ *interp-upd-eq-if-not-mem-vars-fm*
by (*metis vars-fm-vars-branchI*)
ultimately show *?case*
by *auto (metis fun-upd-same)+*
qed

lemma *bexpands-sound*:
assumes *bexpands bs' b*
assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{sa} M \psi$
shows $\exists M. \exists b' \in bs'. \forall \psi \in \text{set } (b' @ b). \text{interp } I_{sa} M \psi$
using *assms*
proof(*induction rule: bexpands.induct*)
case (1 *bs' b*)
with *bexpands-nowit-sound[OF this(1)]* **have** $\exists b' \in bs'. \forall \psi \in \text{set } b'. \text{interp } I_{sa}$
 $M \psi$
by *blast*
with 1 **show** *?case*
by *auto*

```

next
  case (2 t1 t2 x bs b)
  then show ?case using bexpands-wit-sound by metis
qed

```

4.2.4 Upper Bounding the Cardinality of a Branch

lemma *Ex-bexpands-wits-if-in-wits:*

```

assumes wf-branch b
assumes x ∈ wits b
obtains t1 t2 bs b2 b1 where
  expandss b (b2 @ b1) bexpands-wit t1 t2 x bs b1 b2 ∈ bs expandss b1 [last b]
  x ∉ wits b1 wits (b2 @ b1) = insert x (wits b1)
proof –
from assms obtain φ where expandss b [φ]
  unfolding wf-branch-def by blast
then have last b = φ
  by simp
from ⟨expandss b [φ]⟩ ⟨x ∈ wits b⟩ that show ?thesis
  unfolding ⟨last b = φ⟩
proof(induction b [φ] rule: expandss.induct)
  case 1
  then show ?case by simp
next
  case (2 b2 b1)
  with expandss-mono have b1 ≠ []
  by fastforce
  with lexpands-wits-eq[OF ⟨lexpands b2 b1⟩ this] 2 show ?case
  by (metis (no-types, lifting) expandss.intros(2))
next
  case (3 bs - b2)
  then show ?case
proof(induction rule: bexpands.induct)
  case (1 bs b1)
  with expandss-mono have b1 ≠ []
  by fastforce
  with bexpands-nowit-wits-eq[OF ⟨bexpands-nowit bs b1⟩ ⟨b2 ∈ bs⟩ this] 1 show
  ?case
  by (metis bexpands.intros(1) expandss.intros(3))
next
  case (2 t1 t2 y bs b1)
  show ?case
proof(cases x ∈ wits b1)
  case True
  from 2 have expandss (b2 @ b1) b1
  using expandss.intros(3)[OF - 2.prem(1)] bexpands.intros(2)[OF 2.hyps]
  by (metis expandss.intros(1))
  with True 2 show ?thesis
  using expandss-trans by blast

```

```

next
  case False
  from 2 have  $b1 \neq []$ 
    using expandss-mono by fastforce
  with bexpands-witD[OF 2(1)] 2(2-) have  $wits (b2 @ b1) = insert\ y\ (wits\ b1)$ 
    unfolding wits-def
    by (metis 2.hyps bexpands-wit-wits-eq wits-def)
  moreover from  $\langle y \notin vars\text{-}branch\ b1 \rangle$  have  $y \notin wits\ b1$ 
    unfolding wits-def by simp
  moreover from calculation have  $y = x$ 
    using False 2 by simp
  ultimately show ?thesis
    using 2 by (metis expandss.intros(1))
qed
qed
qed
qed

```

lemma *card-wits-ub-if-wf-branch*:

```

assumes wf-branch b
shows  $card\ (wits\ b) \leq (card\ (subterms\ (last\ b)))^2$ 
proof -
  from assms obtain  $\varphi$  where expandss b [ $\varphi$ ]
    unfolding wf-branch-def by blast
  with wf-branch-not-Nil[OF assms] have [simp]:  $last\ b = \varphi$ 
    using expandss-last-eq by force
  have False if card-gt:  $card\ (wits\ b) > (card\ (subterms\ \varphi))^2$ 
  proof -
    define ts where  $ts \equiv (\lambda x. SOME\ t1\text{-}t2. \exists bs\ b2\ b1.$ 
       $expandss\ b\ (b2\ @\ b1) \wedge b2 \in bs \wedge bexpands\text{-}wit\ (fst\ t1\text{-}t2)\ (snd\ t1\text{-}t2)\ x\ bs\ b1$ 
 $\wedge expandss\ b1\ [\varphi])$ 
    from  $\langle expandss\ b\ [\varphi] \rangle \langle last\ b = \varphi \rangle$ 
    have *:  $\exists t1\text{-}t2\ bs\ b2\ b1.$ 
       $expandss\ b\ (b2\ @\ b1) \wedge b2 \in bs \wedge bexpands\text{-}wit\ (fst\ t1\text{-}t2)\ (snd\ t1\text{-}t2)\ x\ bs\ b1$ 
 $\wedge expandss\ b1\ [\varphi]$ 
    if  $x \in wits\ b$  for  $x$ 
    using that Ex-bexpands-wits-if-in-wits[OF  $\langle wf\text{-}branch\ b \rangle$  that] by (metis fst-conv
snd-conv)
    have ts-wd:
       $\exists bs\ b2\ b1. expandss\ b\ (b2\ @\ b1) \wedge b2 \in bs \wedge bexpands\text{-}wit\ t1\ t2\ x\ bs\ b1 \wedge$ 
 $expandss\ b1\ [\varphi]$ 
    if  $ts\ x = (t1, t2)\ x \in wits\ b$  for  $t1\ t2\ x$ 
    using exE-some[OF * that(1)[THEN eq-reflection, symmetric, unfolded ts-def],
OF that(2)]
    by simp
  with  $\langle last\ b = \varphi \rangle \langle expandss\ b\ [\varphi] \rangle$  have in-subterms-fm:
     $t1 \in subterms\ \varphi\ t2 \in subterms\ \varphi$ 
    if  $ts\ x = (t1, t2)\ x \in wits\ b$  for  $t1\ t2\ x$ 

```

```

    using that bexpands-witD
  by (metis expandss-last-eq list.discI)+
have  $\neg$  inj-on ts (wits b)
proof -
  from in-subterms-fm have ts ' wits b  $\subseteq$  subterms  $\varphi \times$  subterms  $\varphi$ 
  by (intro subrelI) (metis imageE mem-Sigma-iff)
  then have card (ts ' wits b)  $\leq$  card (subterms  $\varphi \times$  subterms  $\varphi$ )
  by (intro card-mono) (simp-all add: finite-subterms-fm)
  moreover have card (subterms  $\varphi \times$  subterms  $\varphi$ ) = (card (subterms  $\varphi$ ))2
  unfolding card-cartesian-product by algebra
  ultimately show  $\neg$  inj-on ts (wits b)
  using card-gt by (metis card-image linorder-not-less)
qed

  from  $\langle \neg$  inj-on ts (wits b)  $\rangle$  obtain x t1 t2 xb1 xbs2 xb2 y yb1 ybs2 yb2 where
x-y:
  x  $\neq$  y x  $\in$  wits b y  $\in$  wits b
  expandss xb1 [ $\varphi$ ] bexpands-wit t1 t2 x xbs2 xb1 xb2  $\in$  xbs2 expandss b (xb2 @
xb1)
  expandss yb1 [ $\varphi$ ] bexpands-wit t1 t2 y ybs2 yb1 yb2  $\in$  ybs2 expandss b (yb2 @
yb1)
  unfolding inj-on-def by (metis ts-wd prod.exhaust)
  have xb2  $\neq$  yb2
  using x-y(5)[THEN bexpands-witD(1)] x-y(9)[THEN bexpands-witD(1)] x-y(1,6,10)

  by auto
  moreover from x-y have suffix (xb2 @ xb1) (yb2 @ yb1)  $\vee$  suffix (yb2 @ yb1)
(xb2 @ xb1)
  using expandss-suffix suffix-same-cases by blast
  then have suffix (xb2 @ xb1) yb1  $\vee$  suffix (yb2 @ yb1) xb1
  using x-y(5)[THEN bexpands-witD(1)] x-y(9)[THEN bexpands-witD(1)] x-y(1,6,10)
  by (auto simp: suffix-Cons)
  ultimately show False
  using bexpands-witD(1,5,6)[OF x-y(5)] bexpands-witD(1,5,6)[OF x-y(9)]
x-y(6,10)
  by (auto dest!: set-mono-suffix)
qed
then show ?thesis
  using linorder-not-le  $\langle$  last b =  $\varphi$   $\rangle$  by blast
qed

lemma card-subterms-branch-ub-if-wf-branch:
  assumes wf-branch b
  shows card (subterms b)  $\leq$  card (subterms (last b)) + card (wits b)
  unfolding subterms-branch-eq-if-wf-branch[OF assms, unfolded wits-subterms-def]
  by (simp add: assms card-Un-disjoint card-image-le finite-wits finite-subterms-fm
wits-subterms-last-disjnt)

lemma card-literals-branch-if-wf-branch:

```

```

assumes wf-branch b
shows card {a ∈ set b. is-literal a}
  ≤ 2 * (2 * (card (subterms (last b)) + card (wits b))2)
proof -
  have card {a ∈ set b. is-literal a}
    ≤ card (pset-atoms-branch b) + card (pset-atoms-branch b) (is card ?A ≤ -)
  proof -
    have ?A = {AT a | a. AT a ∈ set b}
      ∪ {AF a | a. AF a ∈ set b} (is - = ?ATs ∪ ?AFs)
    by auto (metis is-literal.elims(2))
  moreover have
    ?ATs ⊆ AT ‘ pset-atoms-branch b ?AFs ⊆ AF ‘ pset-atoms-branch b
  by force+
  moreover from calculation have finite ?ATs finite ?AFs
  by (simp-all add: finite-surj[OF finite-pset-atoms-branch])
  moreover have ?ATs ∩ ?AFs = {}
  by auto
  ultimately show ?thesis
  by (simp add: add-mono card-Un-disjoint finite-pset-atoms-branch surj-card-le)
qed
then have card ?A ≤ 2 * card (pset-atoms-branch b)
  by simp
moreover
have atoms φ ⊆
  case-prod Elem ‘ (subterms φ × subterms φ)
  ∪ case-prod Equal ‘ (subterms φ × subterms φ) for φ :: 'a pset-fm
proof(induction φ)
  case (Atom x)
  then show ?case by (cases x) auto
qed auto
then have pset-atoms-branch b ⊆
  case-prod Elem ‘ (subterms b × subterms b)
  ∪ case-prod Equal ‘ (subterms b × subterms b) (is - ⊆ ?Els ∪ ?Eqs)
  unfolding subterms-branch-def
  by force
have card (pset-atoms-branch b)
  ≤ (card (subterms b))2 + (card (subterms b))2
proof -
  from finite-subterms-branch have finite (subterms b × subterms b)
  using finite-cartesian-product by auto
  then have finite ?Els finite ?Eqs
  by blast+
  moreover have inj-on (case-prod Elem) A inj-on (case-prod Equal) A
  for A :: ('a pset-term × 'a pset-term) set
  unfolding inj-on-def by auto
  ultimately have card ?Els = (card (subterms b))2 card ?Eqs = (card (subterms
b))2
  using card-image[where ?A=subterms b × subterms b] card-cartesian-product
  unfolding power2-eq-square by metis+

```

```

with card-mono[OF - ⟨pset-atoms-branch b ⊆ ?Els ∪ ?Eqs⟩] show ?thesis
  using ⟨finite ?Els⟩ ⟨finite ?Eqs⟩
  by (metis card-Un-le finite-UnI sup.boundedE sup-absorb2)
qed
then have card (pset-atoms-branch b) ≤ 2 * (card (subterms b))2
  by simp
ultimately show ?thesis
  using card-subterms-branch-ub-if-wf-branch[OF assms]
  by (meson dual-order.trans mult-le-mono2 power2-nat-le-eq-le)
qed

lemma lexpands-not-literal-mem-subfms-last:
  defines P ≡ (λb. ∀ψ ∈ set b. ¬ is-literal ψ
    → ψ ∈ subfms (last b) ∨ ψ ∈ Neg ‘ subfms (last b))
  assumes lexpands b' b b ≠ []
  assumes P b
  shows P (b' @ b)
  using assms(2-)
  by (induction b' b rule: lexpands-induct) (fastforce simp: P-def dest: subfmsD)+

lemma bexpands-not-literal-mem-subfms-last:
  defines P ≡ (λb. ∀ψ ∈ set b. ¬ is-literal ψ
    → ψ ∈ subfms (last b) ∨ ψ ∈ Neg ‘ subfms (last b))
  assumes bexpands bs b b' ∈ bs b ≠ []
  assumes P b
  shows P (b' @ b)
  using assms(2-)
proof(induction bs b rule: bexpands.induct)
  case (1 bs' b)
  then show ?case
    apply(induction rule: bexpands-nowit.induct)
    apply(fastforce simp: P-def dest: subfmsD)+
  done
next
  case (2 t1 t2 x bs' b)
  then show ?case
    apply(induction rule: bexpands-wit.induct)
    apply(fastforce simp: P-def dest: subfmsD)+
  done
qed

lemma expandss-not-literal-mem-subfms-last:
  defines P ≡ (λb. ∀ψ ∈ set b. ¬ is-literal ψ
    → ψ ∈ subfms (last b) ∨ ψ ∈ Neg ‘ subfms (last b))
  assumes expandss b' b b ≠ []
  assumes P b
  shows P b'
  using assms(2-)
proof(induction b' b rule: expandss.induct)

```

```

case (2 b3 b2 b1)
then have b2 ≠ []
  using expandss-suffix suffix-bot.extremum-uniqueI by blast
with 2 show ?case
  using lexpands-not-literal-mem-subfms-last unfolding P-def by blast
next
case (3 bs b2 b3 b1)
then have b2 ≠ []
  using expandss-suffix suffix-bot.extremum-uniqueI by blast
with 3 show ?case
  using bexpands-not-literal-mem-subfms-last unfolding P-def by blast
qed simp

```

lemma *card-not-literal-branch-if-wf-branch:*

```

assumes wf-branch b
shows card {φ ∈ set b. ¬ is-literal φ} ≤ 2 * card (subfms (last b))

```

proof –

```

from assms obtain φ where expandss b [φ]

```

```

  unfolding wf-branch-def by blast

```

```

then have [simp]: last b = φ

```

```

  by simp

```

```

have {ψ ∈ set b. ¬ is-literal ψ} ⊆ subfms φ ∪ Neg ‘ subfms φ

```

```

  using expandss-not-literal-mem-subfms-last[OF ‹expandss b [φ]›]

```

```

  by auto

```

```

from card-mono[OF - this] have

```

```

  card {ψ ∈ set b. ¬ is-literal ψ} ≤ card (subfms φ ∪ Neg ‘ subfms φ)

```

```

  using finite-subfms finite-imageI by fast

```

```

also have ... ≤ card (subfms φ) + card (Neg ‘ subfms φ)

```

```

  using card-Un-le by blast

```

```

also have ... ≤ 2 * card (subfms φ)

```

```

  unfolding mult-2 by (simp add: card-image-le finite-subfms)

```

```

finally show ?thesis

```

```

  by simp

```

qed

lemma *card-wf-branch-ub:*

```

assumes wf-branch b

```

```

shows card (set b)

```

```

  ≤ 2 * card (subfms (last b)) + 16 * (card (subterms (last b)))4

```

proof –

```

let ?csts = card (subterms (last b))

```

```

have set b = {ψ ∈ set b. ¬ is-literal ψ} ∪ {ψ ∈ set b. is-literal ψ}

```

```

  by auto

```

```

then have card (set b)

```

```

  = card ({ψ ∈ set b. ¬ is-literal ψ}) + card ({ψ ∈ set b. is-literal ψ})

```

```

  using card-Un-disjoint finite-Un

```

```

  by (metis (no-types, lifting) List.finite-set disjoint-iff mem-Collect-eq)

```

```

also have ... ≤ 2 * card (subfms (last b)) + 4 * (?csts + card (wits b))2

```

```

  using assms card-literals-branch-if-wf-branch card-not-literal-branch-if-wf-branch

```

```

    by fastforce
  also have ... ≤ 2 * card (subfms (last b)) + 4 * (?csts + ?csts2)2
    using assms card-wits-ub-if-wf-branch by auto
  also have ... ≤ 2 * card (subfms (last b)) + 16 * ?csts4
  proof -
    have 1 ≤ ?csts
      using finite-subterms-fm[THEN card-0-eq]
      by (auto intro: Suc-leI)
    then have (?csts + ?csts2)2 = ?csts2 + 2 * ?csts3 + ?csts4
      by algebra
    also have ... ≤ ?csts2 + 2 * ?csts4 + ?csts4
      using power-increasing[OF - <1 ≤ ?csts>] by simp
    also have ... ≤ ?csts4 + 2 * ?csts4 + ?csts4
      using power-increasing[OF - <1 ≤ ?csts>] by simp
    also have ... ≤ 4 * ?csts4
      by simp
    finally show ?thesis
      by simp
  qed
  finally show ?thesis .
  qed

```

4.2.5 The Decision Procedure

```

locale mlss-proc =
  fixes lexpand :: 'a branch ⇒ 'a branch
  assumes lexpands-lexpand:
    ¬ lin-sat b ⇒ lexpands (lexpand b) b ∧ set b ⊂ set (lexpand b @ b)
  fixes bexpand :: 'a branch ⇒ 'a branch set
  assumes bexpands-bexpand:
    ¬ sat b ⇒ lin-sat b ⇒ bexpands (bexpand b) b
begin

function (domintros) mlss-proc-branch :: 'a branch ⇒ bool where
  ¬ lin-sat b
  ⇒ mlss-proc-branch b = mlss-proc-branch (lexpand b @ b)
| [| lin-sat b; bclosed b |] ⇒ mlss-proc-branch b = True
| [| ¬ sat b; bopen b; lin-sat b |]
  ⇒ mlss-proc-branch b = (∀ b' ∈ bexpand b. mlss-proc-branch (b' @ b))
| [| lin-sat b; sat b |] ⇒ mlss-proc-branch b = bclosed b
  by auto

```

lemma mlss-proc-branch-dom-if-wf-branch:

```

assumes wf-branch b
shows mlss-proc-branch-dom b

```

proof -

```

define card-ub :: 'a branch ⇒ nat where
  card-ub ≡ λb. 2 * card (subfms (last b)) + 16 * (card (subterms (last b)))4
from assms show ?thesis

```

```

proof(induction card-ub b - card (set b)
  arbitrary: b rule: less-induct)
  case less
  have less': mlss-proc-branch-dom b' if set b  $\subset$  set b' expandss b' b for b'
  proof -
    note expandss-last-eq[OF  $\langle$ expandss b' b $\rangle$  wf-branch-not-Nil[OF  $\langle$ wf-branch
b $\rangle$ ]]
    then have card-ub b' = card-ub b
      unfolding card-ub-def by simp
    moreover from that  $\langle$ wf-branch b $\rangle$  have wf-branch b'
      by (meson expandss-trans wf-branch-def)
    ultimately have mlss-proc-branch-dom b' if card (set b') > card (set b)
      using less(1)[OF -  $\langle$ wf-branch b' $\rangle$ ] card-wf-branch-ub that
      by (metis (no-types, lifting) card-ub-def diff-less-mono2 order-less-le-trans)
      with that show ?thesis
      by (simp add: psubset-card-mono)
    qed
  then show ?case
  proof(cases sat b)
    case False
    then consider
      b' where  $\neg$  lin-sat b lexpandss b' b set b  $\subset$  set (b' @ b) |
      bs' where lin-sat b  $\neg$  sat b bexpandss bs' b bs'  $\neq$  {}
       $\forall b' \in bs'. set b \subset set (b' @ b)$ 
      unfolding sat-def lin-sat-def
      using bexpandss-strict-mono bexpandss-nonempty
    by (metis (no-types, opaque-lifting) inf-sup-aci(5) psubsetI set-append sup-ge1)
    then show ?thesis
  proof(cases)
    case 1
    with less' show ?thesis
      using mlss-proc-branch.domintros(1)
      by (metis expandss.intros(1,2) lexpandss-expand)
    next
    case 2
    then show ?thesis
      using less' bexpandss-bexpand mlss-proc-branch.domintros(2,3)
      by (metis bexpandss-strict-mono expandss.intros(1,3))
    qed
  qed (use mlss-proc-branch.domintros(4) sat-def in metis)
  qed
qed

```

definition *mlss-proc* :: '*a* *pset-fm* \Rightarrow *bool* **where**
mlss-proc $\varphi \equiv$ *mlss-proc-branch* [φ]

lemma *mlss-proc-branch-complete*:
fixes *b* :: '*a* *branch*
assumes *wf-branch b* $\exists v. v \vdash last b$

```

assumes  $\neg$  mlss-proc-branch b
assumes infinite (UNIV :: 'a set)
shows  $\exists M. \text{interp } I_{sa} M (\text{last } b)$ 
proof –
from mlss-proc-branch-dom-if-wf-branch[OF assms(1)] assms(1,2,3)
show ?thesis
proof(induction rule: mlss-proc-branch.pinduct)
  case (1 b)
  let ?b' = lexpand b
  from 1 lexpands-lexpand have wf-branch (?b' @ b)
    using wf-branch-lexpands by blast
  moreover from 1 lexpands-lexpand have  $\neg$  mlss-proc-branch (?b' @ b)
    by (simp add: mlss-proc-branch.psims)
  ultimately obtain M where interp Isa M (last (?b' @ b))
    using 1 by auto
  with 1 show ?case
    using wf-branch-not-Nil by auto
  next
  case (2 b)
  then show ?case by (simp add: mlss-proc-branch.psims)
  next
  case (3 b)
  let ?bs' = bexpand b
  from 3 bexpands-bexpand obtain b' where b': b' ∈ ?bs'  $\neg$  mlss-proc-branch (b'
  @ b)
    using mlss-proc-branch.psims(3) by metis
  with 3 bexpands-bexpand have wf-branch (b' @ b)
    using wf-branch-expandss[OF  $\langle$ wf-branch b $\rangle$  expandss.intros(3)]
    using expandss.intros(1) by blast
  with 3 b' obtain M where interp Isa M (last (b' @ b))
    by auto
  with 3 show ?case
    by auto
  next
  case (4 b)
  then have bopen b
    by (simp add: mlss-proc-branch.psims)
  interpret open-branch b
    using  $\langle$ wf-branch b $\rangle$   $\langle$  $\exists v. v \vdash \text{last } b \langle$ bopen b $\rangle$   $\langle$ infinite UNIV $\rangle$ 
    by unfold-locales assumption+
  from coherence[OF  $\langle$ sat b $\rangle$  last-in-set] show ?case
    using wf-branch wf-branch-not-Nil by blast
  qed
qed

```

```

lemma mlss-proc-branch-sound:
assumes wf-branch b
assumes  $\forall \psi \in \text{set } b. \text{interp } I_{sa} M \psi$ 
shows  $\neg$  mlss-proc-branch b

```

proof
assume *mlss-proc-branch* *b*
with *mlss-proc-branch-dom-if-wf-branch*[*OF* $\langle \text{wf-branch } b \rangle$]
have $\exists b'. \text{expandss } b' b \wedge (\exists M. \forall \psi \in \text{set } b'. \text{interp } I_{sa} M \psi) \wedge \text{bclosed } b'$
using *assms*
proof(*induction arbitrary: M rule: mlss-proc-branch.pinduct*)
case (1 *b*)
let $?b' = \text{lexpand } b$
from 1 *lexpands-lexpand* $\langle \text{wf-branch } b \rangle$ **have** *wf-branch* ($?b' @ b$)
using *wf-branch-lexpands* **by** *metis*
with 1 *lexpands-sound* *lexpands-lexpand* **obtain** b'' **where**
 $\text{expandss } b'' (?b' @ b) \exists M. \forall \psi \in \text{set } b''. \text{interp } I_{sa} M \psi \text{ bclosed } b''$
by (*fastforce simp: mlss-proc-branch.psimps*)
with 1 *lexpands-lexpand* **show** *?case*
using *expandss-trans expandss.intros(1,2)* **by** *meson*
next
case (3 *b*)
let $?bs' = \text{bexpand } b$
from 3 $\langle \text{wf-branch } b \rangle$ *bexpands-bexpand* **have** *wf-branch-b'*:
 $\text{wf-branch } (b' @ b)$ **if** $b' \in ?bs'$ **for** b'
using *that expandss.intros(3) wf-branch-def* **by** *metis*
from *bexpands-sound bexpands-bexpand* 3 **obtain** $M' b'$ **where**
 $b' \in ?bs' \forall \psi \in \text{set } (b' @ b). \text{interp } I_{sa} M' \psi$
by *metis*
with 3.IH $\langle \text{mlss-proc-branch } b \rangle$ *wf-branch-b'* **obtain** b'' **where**
 $b' \in ?bs' \text{expandss } b'' (b' @ b)$
 $\exists M. \forall \psi \in \text{set } b''. \text{interp } I_{sa} M \psi \text{ bclosed } b''$
using *mlss-proc-branch.psimps(3)[OF 3.hyps(2-4,1)]* **by** *blast*
with 3 *bexpands-bexpand* **show** *?case*
using *expandss-trans expandss.intros(1,3)* **by** *metis*
qed (*use expandss.intros(1) mlss-proc-branch.psimps(4) in* $\langle \text{blast+} \rangle$)
with *bclosed-sound* **show** *False* **by** *blast*
qed

theorem *mlss-proc-complete*:
fixes $\varphi :: 'a \text{ pset-fm}$
assumes $\neg \text{mlss-proc } \varphi$
assumes $\exists v. v \vdash \varphi$
assumes *infinite* (*UNIV* $:: 'a \text{ set}$)
shows $\exists M. \text{interp } I_{sa} M \varphi$
using *assms mlss-proc-branch-complete[of* $[\varphi]$ *]*
unfolding *mlss-proc-def* **by** *simp*

theorem *mlss-proc-sound*:
assumes $\text{interp } I_{sa} M \varphi$
shows $\neg \text{mlss-proc } \varphi$
using *assms mlss-proc-branch-sound[of* $[\varphi]$ *]*
unfolding *mlss-proc-def* **by** *simp*

end

4.3 An Executable Specification of the Procedure

We develop an executable, albeit very inefficient, decision procedure for MLSS. This naive implementation should serve as a proof of concept and a starting point for an efficient implementation.

```
fun subterms-term-list :: 'a pset-term  $\Rightarrow$  'a pset-term list where
  subterms-term-list ( $\emptyset$  n) = [ $\emptyset$  n]
| subterms-term-list (Var i) = [Var i]
| subterms-term-list ( $t1 \sqcup_s t2$ ) = [ $t1 \sqcup_s t2$ ] @ subterms-term-list t1 @ subterms-term-list t2
| subterms-term-list ( $t1 \sqcap_s t2$ ) = [ $t1 \sqcap_s t2$ ] @ subterms-term-list t1 @ subterms-term-list t2
| subterms-term-list ( $t1 -_s t2$ ) = [ $t1 -_s t2$ ] @ subterms-term-list t1 @ subterms-term-list t2
| subterms-term-list (Single t) = [Single t] @ subterms-term-list t
```

```
fun subterms-atom-list :: 'a pset-atom  $\Rightarrow$  'a pset-term list where
  subterms-atom-list ( $t1 \in_s t2$ ) = subterms-term-list t1 @ subterms-term-list t2
| subterms-atom-list ( $t1 =_s t2$ ) = subterms-term-list t1 @ subterms-term-list t2
```

```
fun atoms-list :: 'a pset-fm  $\Rightarrow$  'a pset-atom list where
  atoms-list (Atom a) = [a]
| atoms-list (And p q) = atoms-list p @ atoms-list q
| atoms-list (Or p q) = atoms-list p @ atoms-list q
| atoms-list (Neg p) = atoms-list p
```

```
definition subterms-fm-list :: 'a pset-fm  $\Rightarrow$  'a pset-term list where
  subterms-fm-list  $\varphi \equiv$  concat (map subterms-atom-list (atoms-list  $\varphi$ ))
```

```
definition subterms-branch-list :: 'a branch  $\Rightarrow$  'a pset-term list where
  subterms-branch-list b  $\equiv$  concat (map subterms-fm-list b)
```

```
lemma set-subterms-term-list[simp]:
  set (subterms-term-list t) = subterms t
by (induction t) auto
```

```
lemma set-subterms-atom-list[simp]:
  set (subterms-atom-list t) = subterms t
by (cases t) auto
```

```
lemma set-atoms-list[simp]:
  set (atoms-list  $\varphi$ ) = atoms  $\varphi$ 
by (induction  $\varphi$ ) auto
```

```
lemma set-subterms-fm-list[simp]:
```

$set (subterms\text{-}fm\text{-}list \varphi) = subterms\text{-}fm \varphi$
unfolding $subterms\text{-}fm\text{-}list\text{-}def$ $subterms\text{-}fm\text{-}def$ **by** $simp$

lemma $set\text{-}subterms\text{-}branch\text{-}list[simp]$:
 $set (subterms\text{-}branch\text{-}list b) = subterms b$
unfolding $subterms\text{-}branch\text{-}list\text{-}def$ $subterms\text{-}branch\text{-}def$ **by** $simp$

fun $lexpand\text{-}fm1 :: 'a\ branch \Rightarrow 'a\ pset\text{-}fm \Rightarrow 'a\ branch\ list$ **where**
 $lexpand\text{-}fm1 b (And\ p\ q) = [[p, q]]$
 $| lexpand\text{-}fm1 b (Neg (Or\ p\ q)) = [[Neg\ p, Neg\ q]]$
 $| lexpand\text{-}fm1 b (Or\ p\ q) =$
 $\quad (if\ Neg\ p \in set\ b\ then\ [[q]]\ else\ []) \textcircled{a}$
 $\quad (if\ Neg\ q \in set\ b\ then\ [[p]]\ else\ [])$
 $| lexpand\text{-}fm1 b (Neg (And\ p\ q)) =$
 $\quad (if\ p \in set\ b\ then\ [[Neg\ q]]\ else\ []) \textcircled{a}$
 $\quad (if\ q \in set\ b\ then\ [[Neg\ p]]\ else\ [])$
 $| lexpand\text{-}fm1 b (Neg (Neg\ p)) = [[p]]$
 $| lexpand\text{-}fm1 b - = []$

definition $lexpand\text{-}fm\ b \equiv concat (map (lexpand\text{-}fm1\ b)\ b)$

lemma $lexpand\text{-}fm\text{-}if\text{-}lexpands\text{-}fm$:
 $lexpands\text{-}fm\ b'\ b \Longrightarrow b' \in set (lexpand\text{-}fm\ b)$
apply($induction\ rule: lexpands\text{-}fm.induct$)
apply($force\ simp: lexpand\text{-}fm\text{-}def$)
done

lemma $lexpands\text{-}fm\text{-}if\text{-}lexpand\text{-}fm1$:
 $b' \in set (lexpand\text{-}fm1\ b\ p) \Longrightarrow p \in set\ b \Longrightarrow lexpands\text{-}fm\ b'\ b$
apply($induction\ b\ p\ rule: lexpand\text{-}fm1.induct$)
apply($auto\ simp: lexpands\text{-}fm.intros$)
done

lemma $lexpands\text{-}fm\text{-}if\text{-}lexpand\text{-}fm$:
 $b' \in set (lexpand\text{-}fm\ b) \Longrightarrow lexpands\text{-}fm\ b'\ b$
using $lexpands\text{-}fm\text{-}if\text{-}lexpand\text{-}fm1$ **unfolding** $lexpand\text{-}fm\text{-}def$ **by** $auto$

fun $lexpand\text{-}un1 :: 'a\ branch \Rightarrow 'a\ pset\text{-}fm \Rightarrow 'a\ branch\ list$ **where**
 $lexpand\text{-}un1 b (AF (s \in_s t)) =$
 $\quad [[AF (s \in_s t \sqcup_s t1)]. AF (s' \in_s t1) \leftarrow b, s' = s, t \sqcup_s t1 \in subterms (last\ b)] \textcircled{a}$
 $\quad [[AF (s \in_s t1 \sqcup_s t)]. AF (s' \in_s t1) \leftarrow b, s' = s, t1 \sqcup_s t \in subterms (last\ b)] \textcircled{a}$
 $\quad (case\ t\ of$
 $\quad\quad t1 \sqcup_s t2 \Rightarrow [[AF (s \in_s t1), AF (s \in_s t2)]]$
 $\quad\quad | - \Rightarrow [])$
 $| lexpand\text{-}un1 b (AT (s \in_s t)) =$
 $\quad [[AT (s \in_s t \sqcup_s t2)]. t1 \sqcup_s t2 \leftarrow subterms\text{-}fm\text{-}list (last\ b), t1 = t] \textcircled{a}$
 $\quad [[AT (s \in_s t1 \sqcup_s t)]. t1 \sqcup_s t2 \leftarrow subterms\text{-}fm\text{-}list (last\ b), t2 = t] \textcircled{a}$
 $\quad (case\ t\ of$
 $\quad\quad t1 \sqcup_s t2 \Rightarrow (if\ AF (s \in_s t1) \in set\ b\ then\ [[AT (s \in_s t2)]]\ else\ []) \textcircled{a}$

$(if\ AF\ (s \in_s\ t2) \in\ set\ b\ then\ [[AT\ (s \in_s\ t1)]]\ else\ [])$
 $| - \Rightarrow [])$
 $| \text{lexpand-un1} \ - \ - = []$

definition $\text{lexpand-un } b \equiv \text{concat } (\text{map } (\text{lexpand-un1 } b) b)$

lemma $\text{lexpand-un-if-lexpands-un}$:
 $\text{lexpands-un } b' b \implies b' \in \text{set } (\text{lexpand-un } b)$
apply(*induction rule: lexpands-un.induct*)
apply(*force simp: lexpand-un-def*)
done

lemma $\text{lexpands-un-if-lexpand-un1}$:
 $b' \in \text{set } (\text{lexpand-un1 } b l) \implies l \in \text{set } b \implies \text{lexpands-un } b' b$
apply(*induction b l rule: lexpand-un1.induct*)
apply(*auto simp: lexpands-un.intros*)
done

lemma $\text{lexpands-un-if-lexpand-un}$:
 $b' \in \text{set } (\text{lexpand-un } b) \implies \text{lexpands-un } b' b$
unfolding lexpand-un-def **using** $\text{lexpands-un-if-lexpand-un1}$ **by** *auto*

fun $\text{lexpand-int1} :: 'a\ \text{branch} \Rightarrow 'a\ \text{pset-fm} \Rightarrow 'a\ \text{branch list where}$
 $\text{lexpand-int1 } b\ (AT\ (s \in_s\ t)) =$
 $[[AT\ (s \in_s\ t1 \sqcap_s\ t)].\ AT\ (s' \in_s\ t1) \leftarrow b,\ s' = s,\ t1 \sqcap_s\ t \in \text{subterms } (\text{last } b)]\ @$
 $[[AT\ (s \in_s\ t \sqcap_s\ t2)].\ AT\ (s' \in_s\ t2) \leftarrow b,\ s' = s,\ t \sqcap_s\ t2 \in \text{subterms } (\text{last } b)]\ @$
 $(\text{case } t\ \text{of } t1 \sqcap_s\ t2 \Rightarrow [[AT\ (s \in_s\ t1),\ AT\ (s \in_s\ t2)]]\ | - \Rightarrow [])$
 $| \text{lexpand-int1 } b\ (AF\ (s \in_s\ t)) =$
 $[[AF\ (s \in_s\ t \sqcap_s\ t2)].\ t1 \sqcap_s\ t2 \leftarrow \text{subterms-fm-list } (\text{last } b),\ t1 = t]\ @$
 $[[AF\ (s \in_s\ t1 \sqcap_s\ t)].\ t1 \sqcap_s\ t2 \leftarrow \text{subterms-fm-list } (\text{last } b),\ t2 = t]\ @$
 $(\text{case } t\ \text{of}$
 $t1 \sqcap_s\ t2 \Rightarrow (\text{if } AT\ (s \in_s\ t1) \in \text{set } b\ \text{then } [[AF\ (s \in_s\ t2)]]\ \text{else } [])\ @$
 $(\text{if } AT\ (s \in_s\ t2) \in \text{set } b\ \text{then } [[AF\ (s \in_s\ t1)]]\ \text{else } []))$
 $| - \Rightarrow [])$
 $| \text{lexpand-int1} \ - \ - = []$

definition $\text{lexpand-int } b \equiv \text{concat } (\text{map } (\text{lexpand-int1 } b) b)$

lemma $\text{lexpand-int-if-lexpands-int}$:
 $\text{lexpands-int } b' b \implies b' \in \text{set } (\text{lexpand-int } b)$
apply(*induction rule: lexpands-int.induct*)
apply(*force simp: lexpand-int-def*)
done

lemma $\text{lexpands-int-if-lexpand-int1}$:
 $b' \in \text{set } (\text{lexpand-int1 } b l) \implies l \in \text{set } b \implies \text{lexpands-int } b' b$
apply(*induction b l rule: lexpand-int1.induct*)
apply(*auto simp: lexpands-int.intros*)
done

lemma *lexpands-int-if-lexpand-int*:

$b' \in \text{set } (\text{lexpand-int } b) \implies \text{lexpands-int } b' b$

unfolding *lexpand-int-def* **using** *lexpands-int-if-lexpand-int1* **by** *auto*

fun *lexpand-diff1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**

lexpand-diff1 b (AT (s \in_s t)) =
 [[AT (s \in_s t -_s t2)]. AF (s' \in_s t2) \leftarrow b, s' = s, t -_s t2 \in subterms (last b)] @
 [[AF (s \in_s t1 -_s t)]. AF (s' \in_s t1) \leftarrow b, s' = s, t1 -_s t \in subterms (last b)] @
 [[AF (s \in_s t1 -_s t)]. t1 -_s t2 \leftarrow subterms-fm-list (last b), t2 = t] @
 (case t of t1 -_s t2 \Rightarrow [[AT (s \in_s t1), AF (s \in_s t2)]] | - \Rightarrow [])
 | *lexpand-diff1* b (AF (s \in_s t)) =
 [[AF (s \in_s t -_s t2)]. t1 -_s t2 \leftarrow subterms-fm-list (last b), t1 = t] @
 (case t of
 t1 -_s t2 \Rightarrow (if AT (s \in_s t1) \in set b then [[AT (s \in_s t2)]] else []) @
 (if AF (s \in_s t2) \in set b then [[AF (s \in_s t1)]] else [])
 | - \Rightarrow [])
 | *lexpand-diff1* - - = []

definition *lexpand-diff* b \equiv concat (map (*lexpand-diff1* b) b)

lemma *lexpand-diff-if-lexpands-diff*:

$\text{lexpands-diff } b' b \implies b' \in \text{set } (\text{lexpand-diff } b)$

apply(*induction rule: lexpands-diff.induct*)

apply(*force simp: lexpand-diff-def*)

done

lemma *lexpands-diff-if-lexpand-diff1*:

$b' \in \text{set } (\text{lexpand-diff1 } b) \implies l \in \text{set } b \implies \text{lexpands-diff } b' b$

apply(*induction b l rule: lexpand-diff1.induct*)

apply(*auto simp: lexpands-diff.intros*)

done

lemma *lexpands-diff-if-lexpand-diff*:

$b' \in \text{set } (\text{lexpand-diff } b) \implies \text{lexpands-diff } b' b$

unfolding *lexpand-diff-def* **using** *lexpands-diff-if-lexpand-diff1* **by** *auto*

fun *lexpand-single1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**

lexpand-single1 b (AT (s \in_s Single t)) = [[AT (s =_s t)]]
 | *lexpand-single1* b (AF (s \in_s Single t)) = [[AF (s =_s t)]]
 | *lexpand-single1* - - = []

definition *lexpand-single* b \equiv

[[AT (t1 \in_s Single t1)]. Single t1 \leftarrow subterms-fm-list (last b)] @
 concat (map (*lexpand-single1* b) b)

lemma *lexpand-single-if-lexpands-single*:

$\text{lexpands-single } b' b \implies b' \in \text{set } (\text{lexpand-single } b)$

apply(*induction rule: lexpands-single.induct*)

apply(force simp: lexpand-single-def)+
done

lemma lexpands-single-if-lexpand-single1:
 $b' \in \text{set } (\text{lexpand-single1 } b \ l) \implies l \in \text{set } b \implies \text{lexpands-single } b' \ b$
apply(induction b l rule: lexpand-single1.induct)
apply(auto simp: lexpands-single.intros)
done

lemma lexpands-single-if-lexpand-single:
 $b' \in \text{set } (\text{lexpand-single } b) \implies \text{lexpands-single } b' \ b$
unfolding lexpand-single-def **using** lexpands-single-if-lexpand-single1
by (auto simp: lexpands-single.intros)

fun lexpand-eq1 :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**
lexpand-eq1 b (AT (t1 =_s t2)) =
[[AT (subst-tlvl t1 t2 a)]. AT a \leftarrow b, t1 \in lvl-terms a] @
[[AF (subst-tlvl t1 t2 a)]. AF a \leftarrow b, t1 \in lvl-terms a] @
[[AT (subst-tlvl t2 t1 a)]. AT a \leftarrow b, t2 \in lvl-terms a] @
[[AF (subst-tlvl t2 t1 a)]. AF a \leftarrow b, t2 \in lvl-terms a]
| lexpand-eq1 b - = []

definition lexpand-eq b \equiv
[[AF (s =_s s[^]). AT (s \in _s t) \leftarrow b, AF (s' \in _s t') \leftarrow b, t' = t] @
concat (map (lexpand-eq1 b) b)

lemma lexpand-eq-if-lexpands-eq:
lexpands-eq b' b $\implies b' \in \text{set } (\text{lexpand-eq } b)$
apply(induction rule: lexpands-eq.induct)
apply(force simp: lexpand-eq-def)+
done

lemma lexpands-eq-if-lexpand-eq1:
 $b' \in \text{set } (\text{lexpand-eq1 } b \ l) \implies l \in \text{set } b \implies \text{lexpands-eq } b' \ b$
apply(induction b l rule: lexpand-eq1.induct)
apply(auto simp: lexpands-eq.intros)
done

lemma lexpands-eq-if-lexpand-eq:
 $b' \in \text{set } (\text{lexpand-eq } b) \implies \text{lexpands-eq } b' \ b$
unfolding lexpand-eq-def **using** lexpands-eq-if-lexpand-eq1
by (auto simp: lexpands-eq.intros)

definition lexpand b \equiv
lexpand-fm b @
lexpand-un b @ lexpand-int b @ lexpand-diff b @
lexpand-single b @ lexpand-eq b

lemma lexpand-if-lexpands:

$lexpands\ b'\ b \implies b' \in set\ (lexpand\ b)$
apply(*induction rule: lexpands.induct*)
unfolding *lexpand-def*
using *lexpand-fm-if-lexpands-fm*
using *lexpand-un-if-lexpands-un lexpand-int-if-lexpands-int lexpand-diff-if-lexpands-diff*
using *lexpand-single-if-lexpands-single lexpand-eq-if-lexpands-eq*
by *fastforce+*

lemma *lexpands-if-lexpand*:
 $b' \in set\ (lexpand\ b) \implies lexpands\ b'\ b$
unfolding *lexpand-def*
using *lexpands-fm-if-lexpand-fm*
using *lexpands-un-if-lexpand-un lexpands-int-if-lexpand-int lexpands-diff-if-lexpand-diff*
using *lexpands-single-if-lexpand-single lexpands-eq-if-lexpand-eq*
using *lexpands.intros* **by** *fastforce*

fun *bexpand-nowit1* :: '*a* *branch* \implies '*a* *pset-fm* \implies '*a* *branch list list* **where**
bexpand-nowit1 *b* (*Or* *p* *q*) =
 (*if* $p \notin set\ b \wedge Neg\ p \notin set\ b$ then $[[[p], [Neg\ p]]]$ else $[]$)
| *bexpand-nowit1* *b* (*Neg* (*And* *p* *q*)) =
 (*if* $Neg\ p \notin set\ b \wedge p \notin set\ b$ then $[[[Neg\ p], [p]]]$ else $[]$)
| *bexpand-nowit1* *b* (*AT* (*s* \in_s *t*)) =
 $[[[AT\ (s \in_s\ t2)], [AF\ (s \in_s\ t2)]]]$. $t' \sqcap_s\ t2 \leftarrow subterms-fm-list\ (last\ b)$, $t' = t$,
 $AT\ (s \in_s\ t2) \notin set\ b$, $AF\ (s \in_s\ t2) \notin set\ b$ @
 $[[[AT\ (s \in_s\ t2)], [AF\ (s \in_s\ t2)]]]$. $t' -_s\ t2 \leftarrow subterms-fm-list\ (last\ b)$, $t' = t$,
 $AT\ (s \in_s\ t2) \notin set\ b$, $AF\ (s \in_s\ t2) \notin set\ b$ @
 (*case* *t* *of*
 t1 \sqcup_s *t2* \implies
 (*if* $t1 \sqcup_s\ t2 \in subterms\ (last\ b) \wedge AT\ (s \in_s\ t1) \notin set\ b \wedge AF\ (s \in_s\ t1) \notin$
set *b*
 then $[[[AT\ (s \in_s\ t1)], [AF\ (s \in_s\ t1)]]]$ else $[]$)
 | - $\implies []$)
| *bexpand-nowit1* *b* - = $[]$

definition *bexpand-nowit* *b* $\equiv concat\ (map\ (bexpand-nowit1\ b)\ b)$

lemma *bexpand-nowit-if-bexpands-nowit*:
 $bexpands-nowit\ bs'\ b \implies bs' \in set\ 'set\ (bexpand-nowit\ b)$
apply(*induction rule: bexpands-nowit.induct*)
apply(*force simp: bexpand-nowit-def*)
done

lemma *bexpands-nowit-if-bexpand-nowit1*:
 $bs' \in set\ 'set\ (bexpand-nowit1\ b\ l) \implies l \in set\ b \implies bexpands-nowit\ bs'\ b$
apply(*induction b l rule: bexpand-nowit1.induct*)
apply(*auto simp: bexpands-nowit.intros*)
done

lemma *bexpands-nowit-if-bexpand-nowit*:

$bs' \in \text{set } \text{'set } (\text{bexpand-nowit } b) \implies \text{bexpands-nowit } bs' b$
unfolding *bexpand-nowit-def* **using** *bexpands-nowit-if-bexpand-nowit1*
by (*auto simp: bexpands-nowit.intros*)

definition *name-subterm* $\varphi \equiv \text{index } (\text{subterms-fm-list } \varphi)$

lemma *inj-on-name-subterm-subterms*:
inj-on (*name-subterm* φ) (*subterms* φ)
unfolding *name-subterm-def*
by (*intro inj-on-index2*) *simp*

abbreviation *solve-constraints* $\varphi \equiv$
MLSS-Suc-Theory.solve (*MLSS-Suc-Theory.elim-NEq-Zero* (*constrs-fm* (*name-subterm*
 φ) φ))

definition *urelem-code* $\varphi t \equiv$
case solve-constraints φ *of*
Some ss \Rightarrow *MLSS-Suc-Theory.assign* *ss* (*name-subterm* φt) = 0
| *None* \Rightarrow *False*

lemma *urelem-code-if-mem-subterms*:
assumes $t \in \text{subterms } \varphi$
shows *urelem* $\varphi t \longleftrightarrow \text{urelem-code } \varphi t$
proof –
note *urelem-iff-assign-eq-0[OF - assms]* *not-types-fm-if-solve-eq-None*
note *solve-correct = this[OF inj-on-name-subterm-subterms]*
then show *?thesis*
unfolding *urelem-def urelem-code-def*
by (*auto split: option.splits*)
qed

fun *bexpand-wit1* :: (*'a::fresh0*) *branch* \Rightarrow *'a pset-fm* \Rightarrow *'a branch list list* **where**
bexpand-wit1 b (*AF* ($t1 =_s t2$)) =
($\forall t \in \text{set } b. \text{case } t \text{ of } AT (x \in_s t1') \Rightarrow t1' = t1 \longrightarrow AF (x \in_s t2) \notin \text{set } b$ |
- \Rightarrow *True*) \wedge
($\forall t \in \text{set } b. \text{case } t \text{ of } AT (x \in_s t2') \Rightarrow t2' = t2 \longrightarrow AF (x \in_s t1) \notin \text{set } b$ |
- \Rightarrow *True*) \wedge
 $\neg \text{urelem-code } (\text{last } b) t1 \wedge \neg \text{urelem-code } (\text{last } b) t2$
then
(*let* $x = \text{fresh0 } (\text{vars } b)$
in [[*AT* (*Var* $x \in_s t1$), *AF* (*Var* $x \in_s t2$)],
[*AT* (*Var* $x \in_s t2$), *AF* (*Var* $x \in_s t1$)]]])
else [])
| *bexpand-wit1* b - = []

definition *bexpand-wit* $b \equiv \text{concat } (\text{map } (\text{bexpand-wit1 } b) b)$

lemma *Not-Ex-wit-code*:

$(\nexists x. AT(x \in_s t1) \in set\ b \wedge AF(x \in_s t2) \in set\ b)$
 $\longleftrightarrow (\forall fm \in set\ b. case\ fm\ of$
 $\quad AT(x \in_s t') \Rightarrow t' = t1 \longrightarrow AF(x \in_s t2) \notin set\ b$
 $\quad | _ \Rightarrow True)$
by (*auto split: fm.splits pset-atom.splits*)

lemma *bexpand-wit1-if-bexpands-wit*:
assumes *bexpands-wit t1 t2 (fresh0 (vars b)) bs' b*
shows *bs' \in set ' set (bexpand-wit1 b (AF (t1 =_s t2)))*
proof –
from *bexpands-witD[OF assms] show ?thesis*
by (*simp add: Let-def urelem-code-if-mem-subterms Not-Ex-wit-code[symmetric]*)
qed

lemma *bexpand-wit-if-bexpands-wit*:
assumes *bexpands-wit t1 t2 (fresh0 (vars b)) bs' b*
shows *bs' \in set ' set (bexpand-wit b)*
using *assms(1)[THEN bexpand-wit1-if-bexpands-wit] bexpands-witD(2)[OF assms(1)]*
unfolding *bexpand-wit-def*
by (*auto simp del: bexpand-wit1.simps(1)*)

lemma *bexpands-wit-if-bexpand-wit1*:
 $b' \in set ' set (bexpand-wit1\ b\ l) \implies l \in set\ b \implies (\exists t1\ t2\ x. bexpands-wit\ t1\ t2\ x\ b'\ b)$
proof(*induction b l rule: bexpand-wit1.induct*)
case (*1 b t1 t2*)
show *?case*
apply(*rule exI[where ?x=t1], rule exI[where ?x=t2],*
 $rule\ exI[where\ ?x=fresh0\ (vars\ b)]$)
using *1*
by (*auto simp: Let-def bexpands-wit.simps finite-vars-branch[THEN fresh0-notIn]*
 $Not-Ex-wit-code[symmetric]\ urelem-code-if-mem-subterms$)
qed *auto*

lemma *bexpands-wit-if-bexpand-wit*:
 $bs' \in set ' set (bexpand-wit\ b) \implies (\exists t1\ t2\ x. bexpands-wit\ t1\ t2\ x\ bs'\ b)$
proof –
assume *bs' \in set ' set (bexpand-wit b)*
then obtain l where *bs' \in set ' set (bexpand-wit1 b l) l \in set b*
unfolding *bexpand-wit-def* **by** *auto*
from *bexpands-wit-if-bexpand-wit1[OF this] show ?thesis .*
qed

definition *bexpand b \equiv bexpand-nowit b @ bexpand-wit b*

lemma *bexpands-if-bexpand*:
 $bs' \in set ' set (bexpand\ b) \implies bexpands\ bs'\ b$
unfolding *bexpand-def*
using *bexpands-nowit-if-bexpand-nowit bexpands-wit-if-bexpand-wit*

by (metis *bexpands.intros UnE image-Un set-append*)

lemma *Not-bexpands-if-bexpand-empty*:

assumes *bexpand b = []*

shows \neg *bexpands bs' b*

proof

assume *bexpands bs' b*

then show *False*

using *assms*

proof (*induction rule: bexpands.induct*)

case (1 *bs' b*)

with *bexpand-nowit-if-bexpands-nowit[OF this(1)]* **show** *?case*

unfolding *bexpand-def* **by** *simp*

next

case (2 *t1 t2 x bs' b*)

note *fresh-notIn[OF finite-vars-branch, of b]*

with 2 **obtain** *bs''* **where** *bexpands-wit t1 t2 (fresh0 (vars b)) bs'' b*

unfolding *fresh0-def* **by** (*auto simp: bexpands-wit.simps*)

from 2 *bexpand-wit-if-bexpands-wit[OF this]* **show** *?case*

by (*simp add: bexpand-def*)

qed

qed

lemma *lin-sat-code*:

lin-sat b \longleftrightarrow *filter* ($\lambda b'. \neg$ *set b' \subseteq set b*) (*lexpand b*) = []

unfolding *lin-sat-def*

using *lexpand-if-lexpands lexpands-if-lexpand*

by (*force simp: filter-empty-conv*)

lemma *sat-code*:

sat b \longleftrightarrow *lin-sat b* \wedge *bexpand b* = []

using *Not-bexpands-if-bexpand-empty bexpands-if-bexpand*

unfolding *sat-def*

by (*metis imageI list.set-intros(1) list-exhaust2*)

fun *bclosed-code1* :: '*a* *branch* \Rightarrow '*a* *pset-fm* \Rightarrow *bool* **where**

bclosed-code1 b (*Neg φ*) \longleftrightarrow

$\varphi \in$ *set b* \vee

(*case φ of Atom (t1 =_s t2) \Rightarrow t1 = t2 | - \Rightarrow False*)

| *bclosed-code1 b* (*AT (- \in_s \emptyset -)*) \longleftrightarrow *True*

| *bclosed-code1 - -* \longleftrightarrow *False*

definition *bclosed-code b* \equiv ($\exists t \in$ *set b*. *bclosed-code1 b t*)

lemma *bclosed-code-if-bclosed*:

assumes *bclosed b wf-branch b v \vdash last b*

shows *bclosed-code b*

using *assms*

proof(*induction rule: bclosed.induct*)

```

case (contr  $\varphi$  b)
then have bclosed-code1 b (Neg  $\varphi$ )
  by auto
with contr show ?case
  unfolding bclosed-code-def by blast
next
case (memEmpty t n b)
then have bclosed-code1 b (AT ( $t \in_s \emptyset n$ ))
  by auto
with memEmpty show ?case
  unfolding bclosed-code-def by blast
next
case (negSelf t b)
then have bclosed-code1 b (AF ( $t =_s t$ ))
  by auto
with negSelf show ?case
  unfolding bclosed-code-def by blast
next
case (memberCycle cs b)
then show ?case
  by (auto simp: bclosed-code-def dest: no-member-cycle-if-types-last)
qed

```

lemma *bclosed-if-bclosed-code1*:
 $bclosed-code1\ b\ l \implies l \in set\ b \implies bclosed\ b$
by (*induction rule: bclosed-code1.induct*)
(auto simp: bclosed.intros split: fm.splits pset-atom.splits)

lemma *bclosed-if-bclosed-code*:
 $bclosed-code\ b \implies bclosed\ b$
unfolding *bclosed-code-def* **using** *bclosed-if-bclosed-code1* **by** *blast*

lemma *bclosed-code*:
assumes *wf-branch* *b v* $\vdash last\ b$
shows $bclosed\ b \longleftrightarrow bclosed-code\ b$
using *assms bclosed-if-bclosed-code bclosed-code-if-bclosed*
by *blast*

definition *lexpand-safe* $b \equiv$
case filter ($\lambda b'. \neg set\ b' \subseteq set\ b$) (*lexpand* *b*) *of*
 $b' \# bs' \Rightarrow b'$
 $|\ \square \Rightarrow \square$

lemma *lexpands-lexpand-safe*:
 $\neg lin-sat\ b \implies lexpands\ (lexpand-safe\ b)\ b \wedge set\ b \subset set\ (lexpand-safe\ b\ @\ b)$
unfolding *lexpand-safe-def*
by (*auto simp: lin-sat-code intro!: lexpands-if-lexpand dest: filter-eq-ConsD split: list.splits*)

lemma *wf-branch-lexpand-safe*:
assumes *wf-branch b*
shows *wf-branch (lexpand-safe b @ b)*
proof –
from *assms* **have** *wf-branch (lexpand-safe b @ b)* **if** \neg *lin-sat b*
using *that lexpands-lexpand-safe wf-branch-lexpands* **by** *metis*
moreover **have** *wf-branch (lexpand-safe b @ b)* **if** *lin-sat b*
using *assms that[unfolded lin-sat-code]*
unfolding *lexpand-safe-def* **by** *simp*
ultimately show *?thesis*
by *blast*
qed

definition *bexpand-safe b* \equiv
case bexpand b of
bs' # bss' \Rightarrow bs'
| [] \Rightarrow [[]]

lemma *bexpands-bexpand-safe*:
 \neg *sat b* \Longrightarrow *lin-sat b* \Longrightarrow *bexpands (set (bexpand-safe b)) b*
unfolding *bexpand-safe-def*
by (*auto simp: sat-code bexpands-if-bexpand split: list.splits*)

lemma *wf-branch-bexpand-safe*:
assumes *wf-branch b*
shows $\forall b' \in \text{set } (bexpand\text{-safe } b).$ *wf-branch (b' @ b)*
proof –
note *wf-branch-expandss[OF assms expandss.intros(3), OF bexpands-if-bexpand]*
with *assms* **show** *?thesis*
unfolding *bexpand-safe-def*
by (*simp split: list.splits (metis expandss.intros(1) image-iff list.set-intros(1))*)
qed

interpretation *mlss-naive: mlss-proc lexpand-safe set o bexpand-safe*
apply(*unfold-locales*)
using *lexpands-lexpand-safe bexpands-bexpand-safe* **by** *auto*

lemma *types-pset-fm-code*:
 $(\exists v. v \vdash \varphi) \iff \text{solve-constraints } \varphi \neq \text{None}$
using *not-types-fm-if-solve-eq-None types-pset-fm-assign-solve*
by (*meson inj-on-name-subterm-subterms not-Some-eq*)

fun *foldl-option* **where**
foldl-option f a [] = Some a
| foldl-option f - (None # -) = None
| foldl-option f a (Some x # xs) = foldl-option f (f a x) xs

lemma *monotone-fold-option-conj[partial-function-mono]*:
monotone (list-all2 option-ord) option-ord (foldl-option f a)

```

proof
  fix xs ys :: 'a option list
  assume list-all2 option-ord xs ys
  then show option-ord (foldl-option f a xs) (foldl-option f a ys)
  proof(induction xs ys arbitrary: a rule: list-all2-induct)
    case Nil
    then show ?case by (simp add: option.leq-refl)
  next
    case (Cons xo xos yo yos)
    then consider
      xo = None yo = None
    | y where xo = None yo = Some y
    | x y where xo = Some x yo = Some y
    by (metis flat-ord-def option.exhaust)
    then show ?case
    using Cons
    by cases (simp-all add: option.leq-refl flat-ord-def)
  qed
qed

```

```

lemma monotone-map[partial-function-mono]:
  assumes monotone (list-all2 option-ord) ordb B
  shows monotone option.le-fun ordb (λh. B (map h xs))
  using assms
  by (simp add: fun-ord-def list-all2-conv-all-nth monotone-on-def)

```

```

partial-function (option) mlss-proc-branch-partial
  :: ('a::fresh0) branch ⇒ bool option where
  mlss-proc-branch-partial b =
    (if ¬ lin-sat b then mlss-proc-branch-partial (lexpand-safe b @ b)
    else if bclosed-code b then Some True
    else if ¬ sat b then
      foldl-option (∧) True (map mlss-proc-branch-partial (map (λb'. b' @ b)
    (bcxand-safe b)))
    else Some (bclosed-code b))

```

```

lemma mlss-proc-branch-partial-eq:
  assumes wf-branch b v ⊢ last b
  shows mlss-proc-branch-partial b = Some (mlss-naive.mlss-proc-branch b)
    (is ?mlss-part b = Some (?mlss b))
  using mlss-naive.mlss-proc-branch-dom-if-wf-branch[OF assms(1)] assms
  proof(induction rule: mlss-naive.mlss-proc-branch.pinduct)
    case (1 b)
    then have ?mlss-part (lexpand-safe b @ b)
      = Some (mlss-naive.mlss-proc-branch (lexpand-safe b @ b))
    using wf-branch-lexpand-safe[OF <wf-branch b>] by fastforce
  with 1 show ?case
    by (subst mlss-proc-branch-partial.simps)
      (auto simp: mlss-naive.mlss-proc-branch.psimps)

```

```

next
  case (2 b)
  then show ?case
    by (subst mlss-proc-branch-partial.simps)
      (auto simp: mlss-naive.mlss-proc-branch.psimps bclosed-code)
next
  case (3 b)
  then have ?mlss-part (b' @ b) = Some (?mlss (b' @ b))
    if b' ∈ set (bexpand-safe b) for b'
    using that wf-branch-bexpand-safe[OF ‹wf-branch b›] by fastforce
  then have map ?mlss-part (map (λb'. b' @ b) (bexpand-safe b))
    = map Some (map (λb'. (?mlss (b' @ b))) (bexpand-safe b))
    by simp
  moreover have foldl-option (λ) a (map Some xs) = Some (a ∧ (∀ x ∈ set xs.
x)) for a xs
    by (induction xs arbitrary: a) auto
  moreover have foldl-option-eq:
    foldl-option (λ) True (map ?mlss-part (map (λb'. b' @ b) (bexpand-safe b)))
    = Some (∀ b' ∈ set (bexpand-safe b). ?mlss (b' @ b))
    unfolding calculation by (auto simp: comp-def)
  from 3 show ?case
    by (subst mlss-proc-branch-partial.simps, subst foldl-option-eq)
      (simp add: bclosed-code mlss-naive.mlss-proc-branch.psimps(3))
next
  case (4 b)
  then show ?case
    by (subst mlss-proc-branch-partial.simps)
      (auto simp: mlss-naive.mlss-proc-branch.psimps bclosed-code)
qed

```

definition *mlss-proc-partial* ($\varphi :: \text{nat pset-fm}$) \equiv
if solve-constraints $\varphi = \text{None}$ then None else mlss-proc-branch-partial [φ]

lemma *mlss-proc-partial-eq-None*:
mlss-proc-partial $\varphi = \text{None} \implies (\nexists v. v \vdash \varphi)$
unfolding *mlss-proc-partial-def*
using *types-pset-fm-code mlss-proc-branch-partial-eq wf-branch-singleton*
by (*metis last.simps option.discI*)

lemma *mlss-proc-partial-complete*:
assumes *mlss-proc-partial* $\varphi = \text{Some False}$
shows $\exists M. \text{interp } I_{sa} M \varphi$
proof –
from *assms* **have** $\exists v. v \vdash \varphi$
unfolding *mlss-proc-partial-def* **using** *types-pset-fm-code* **by** *force*
moreover **have** $\neg \text{mlss-naive.mlss-proc } \varphi$
using *assms* $\langle \exists v. v \vdash \varphi \rangle$ *mlss-proc-branch-partial-eq calculation wf-branch-singleton*
unfolding *mlss-naive.mlss-proc-def mlss-proc-partial-def*
by (*metis last.simps option.discI option.inject*)

```

ultimately show ?thesis
  using mlss-naive.mlss-proc-complete by blast
qed

lemma mlss-proc-partial-sound:
  assumes mlss-proc-partial  $\varphi = \text{Some True}$ 
  shows  $\neg \text{interp } I_{sa} M \varphi$ 
proof -
  from assms have  $\exists v. v \vdash \varphi$ 
  unfolding mlss-proc-partial-def using types-pset-fm-code by force
  moreover have mlss-naive.mlss-proc  $\varphi$ 
  using assms  $\langle \exists v. v \vdash \varphi \rangle$  mlss-proc-branch-partial-eq calculation wf-branch-singleton
  unfolding mlss-naive.mlss-proc-def mlss-proc-partial-def
  by (metis last.simps option.discI option.inject)
  ultimately show ?thesis
  using mlss-naive.mlss-proc-sound by blast
qed

declare lin-sat-code[code] sat-code[code]
declare mlss-proc-branch-partial.simps[code]
code-identifier
  code-module MLSS-Calculus  $\rightarrow$  (SML) MLSS-Proc-Code
  | code-module MLSS-Proc  $\rightarrow$  (SML) MLSS-Proc-Code
export-code mlss-proc-partial in SML

```

Bibliography

- [1] D. Cantone and C. G. Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics, Selected Papers*, volume 1761 of *Lecture Notes in Computer Science*, pages 126–136. Springer, 1998.