

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	8
1.2.5	Non-Emptiness of Subterms	9
2	A Tableau Calculus for MLSS	10
2.1	Typing Rules	10
2.2	The Calculus	11
2.2.1	Closedness	11
2.2.2	Linear Expansion Rules	11
2.2.3	Branching Expansion Rules	15
2.2.4	Expansion Closure	16
2.2.5	Well-Formed Branch	17
3	A Type Checker for MLSS	18
3.1	Solver for the Theory of the Successor Function	18
3.2	Typing and Branch Expansion	24
3.3	Typing the Urelements	28
4	Deciding MLSS	33
4.1	The Realisation Function	33
4.2	A Decision Procedure for MLSS	36
4.2.1	Basic Definitions	36
4.2.2	Completeness of the Calculus	40
4.2.3	Soundness of the Calculus	50
4.2.4	Upper Bounding the Cardinality of a Branch	52
4.2.5	The Decision Procedure	53
4.3	An Executable Specification of the Procedure	54

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*
 <*proof*>

lemma *Atoms-mono*: *A* \subseteq *B* \implies *Atoms* *A* \subseteq *Atoms* *B*
 <*proof*>

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

<proof>

lemma *vars-fmI*:

$x \in$ *vars* $p \implies x \in$ *vars* (*And* p q)

$x \in$ *vars* $q \implies x \in$ *vars* (*And* p q)

$x \in$ *vars* $p \implies x \in$ *vars* (*Or* p q)

$x \in$ *vars* $q \implies x \in$ *vars* (*Or* p q)

$x \in$ *vars* $p \implies x \in$ *vars* (*Neg* p)

<proof>

lemma *vars-branch-simps*:

vars [] = {}

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

<proof>

lemma *vars-branch-append*:

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

<proof>

lemma *vars-fm-vars-branchI*:

$\varphi \in$ set $b \implies x \in$ *vars-fm* $\varphi \implies x \in$ *vars-branch* b

<proof>

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 $b \equiv \bigcup (subfms\ 'set\ b)$

lemma *subfms-branch-simps*:
subfms-branch $[] = \{\}$
subfms-branch $(x\ \#\ xs) = subfms\ x \cup subfms-branch\ xs$
<proof>

lemma *subfms-refl[simp]*: $p \in subfms\ p$
<proof>

lemma *subfmsI*:
 $a \in subfms\ p \Longrightarrow a \in subfms\ (And\ p\ q)$
 $a \in subfms\ q \Longrightarrow a \in subfms\ (And\ p\ q)$
 $a \in subfms\ p \Longrightarrow a \in subfms\ (Or\ p\ q)$
 $a \in subfms\ q \Longrightarrow a \in subfms\ (Or\ p\ q)$
 $a \in subfms\ p \Longrightarrow a \in subfms\ (Neg\ p)$
<proof>

lemma *subfms-trans*: $q \in subfms\ p \Longrightarrow p \in subfms\ r \Longrightarrow q \in subfms\ r$
<proof>

lemma *subfmsD*:
 $And\ p\ q \in subfms\ \varphi \Longrightarrow p \in subfms\ \varphi$
 $And\ p\ q \in subfms\ \varphi \Longrightarrow q \in subfms\ \varphi$
 $Or\ p\ q \in subfms\ \varphi \Longrightarrow p \in subfms\ \varphi$
 $Or\ p\ q \in subfms\ \varphi \Longrightarrow q \in subfms\ \varphi$
 $Neg\ p \in subfms\ \varphi \Longrightarrow p \in subfms\ \varphi$
<proof>

Subterms

fun *subterms-term* :: 'a pset-term \Rightarrow 'a pset-term set **where**
subterms-term $(\emptyset\ n) = \{\emptyset\ n\}$
| *subterms-term* $(Var\ i) = \{Var\ i\}$
| *subterms-term* $(t1\ \sqcup_s\ t2) = \{t1\ \sqcup_s\ t2\} \cup subterms-term\ t1 \cup subterms-term\ t2$
| *subterms-term* $(t1\ \sqcap_s\ t2) = \{t1\ \sqcap_s\ t2\} \cup subterms-term\ t1 \cup subterms-term\ t2$
| *subterms-term* $(t1\ -_s\ t2) = \{t1\ -_s\ t2\} \cup subterms-term\ t1 \cup subterms-term\ t2$
| *subterms-term* $(Single\ t) = \{Single\ t\} \cup subterms-term\ t$

fun *subterms-atom* :: 'a pset-atom \Rightarrow 'a pset-term set **where**
subterms-atom $(t1\ \in_s\ t2) = subterms-term\ t1 \cup subterms-term\ t2$
| *subterms-atom* $(t1\ =_s\ t2) = subterms-term\ t1 \cup subterms-term\ t2$

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

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

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

adhoc-overloading

subterms \equiv *subterms-term* **and**
subterms \equiv *subterms-atom* **and**
subterms \equiv *subterms-fm* **and**
subterms \equiv *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
 <proof>

lemma *subterms-branch-simps*:

subterms [] = {}
subterms (x # xs) = *subterms* x \cup *subterms* xs
 <proof>

lemma *subterms-refl*[simp]:

t \in *subterms* t
 <proof>

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

s \in *subterms-term* t \implies t \in *subterms-term* v \implies s \in *subterms-term* v
 <proof>

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

s \in *subterms-term* t \implies t \in *subterms-atom* v \implies s \in *subterms-atom* v
 <proof>

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

s \in *subterms-term* t \implies t \in *subterms-fm* $\varphi \implies$ s \in *subterms-fm* φ
 <proof>

lemma *subterms-fmD*:

t1 \sqcup_s t2 \in *subterms-fm* $\varphi \implies$ t1 \in *subterms-fm* φ
 t1 \sqcup_s t2 \in *subterms-fm* $\varphi \implies$ t2 \in *subterms-fm* φ
 t1 \sqcap_s t2 \in *subterms-fm* $\varphi \implies$ t1 \in *subterms-fm* φ
 t1 \sqcap_s t2 \in *subterms-fm* $\varphi \implies$ t2 \in *subterms-fm* φ
 t1 $-_s$ t2 \in *subterms-fm* $\varphi \implies$ t1 \in *subterms-fm* φ
 t1 $-_s$ t2 \in *subterms-fm* $\varphi \implies$ t2 \in *subterms-fm* φ
 Single t \in *subterms-fm* $\varphi \implies$ t \in *subterms-fm* φ
 <proof>

lemma *subterms-branchD*:

$t1 \sqcup_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 \sqcup_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
 $t1 \sqcap_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 \sqcap_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
 $t1 -_s t2 \in \text{subterms-branch } b \implies t1 \in \text{subterms-branch } b$
 $t1 -_s t2 \in \text{subterms-branch } b \implies t2 \in \text{subterms-branch } b$
 $\text{Single } t \in \text{subterms-branch } b \implies t \in \text{subterms-branch } b$
(proof)

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

$s \in \text{subterms-term } t \implies t \in \text{subterms-branch } b \implies s \in \text{subterms-branch } b$
(proof)

lemma *AT-mem-subterms-branchD*:

assumes $AT (s \in_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
(proof)

lemma *AF-mem-subterms-branchD*:

assumes $AF (s \in_s t) \in \text{set } b$
shows $s \in \text{subterms } b \implies t \in \text{subterms } b$
(proof)

lemma *AT-eq-subterms-branchD*:

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

lemma *AF-eq-subterms-branchD*:

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

Interactions between Subterms and Subformulae

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

$\bigcup (\text{vars-term } ' \text{subterms } t) = \text{vars-term } t$
(proof)

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

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

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

$\bigcup (\text{vars-term } ' \text{subterms-branch } b) = \text{vars-branch } b$
(proof)

lemma *subs-vars-branch-if-subs-subterms-branch*:

$subterms\text{-}branch\ b1 \subseteq subterms\text{-}branch\ b2 \implies vars\text{-}branch\ b1 \subseteq vars\text{-}branch\ b2$
 ⟨proof⟩

lemma *subterms-branch-eq-if-vars-branch-eq:*

$subterms\text{-}branch\ b1 = subterms\text{-}branch\ b2 \implies vars\text{-}branch\ b1 = vars\text{-}branch\ b2$
 ⟨proof⟩

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

$x \in vars\text{-}term\ s \implies s \in subterms\text{-}term\ t \implies x \in vars\text{-}term\ t$
 ⟨proof⟩

lemma *mem-vars-fm-if-mem-subterms-fm:*

$x \in vars\text{-}term\ s \implies s \in subterms\text{-}fm\ \varphi \implies x \in vars\text{-}fm\ \varphi$
 ⟨proof⟩

lemma *vars-term-subs-subterms-term:*

$v \in vars\text{-}term\ t \implies Var\ v \in subterms\text{-}term\ t$
 ⟨proof⟩

lemma *vars-atom-subs-subterms-atom:*

$v \in vars\text{-}atom\ a \implies Var\ v \in subterms\text{-}atom\ a$
 ⟨proof⟩

lemma *vars-fm-subs-subterms-fm:*

$v \in vars\text{-}fm\ \varphi \implies Var\ v \in subterms\text{-}fm\ \varphi$
 ⟨proof⟩

lemma *vars-branch-subs-subterms-branch:*

$Var\ v \in vars\text{-}branch\ b \subseteq subterms\text{-}branch\ b$
 ⟨proof⟩

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

$Atom\ a \in set\ b \implies x \in subterms\text{-}term\ s \implies s \in subterms\text{-}atom\ a \implies x \in subterms\text{-}branch\ b$
 ⟨proof⟩

lemma *subterms-branch-subterms-subterms-fm-trans:*

$b \neq [] \implies x \in subterms\text{-}term\ t \implies t \in subterms\text{-}fm\ (last\ b) \implies x \in subterms\text{-}branch\ b$
 ⟨proof⟩

Set Atoms in a Branch

abbreviation *pset-atoms-branch* :: 'a fm list \Rightarrow 'a set **where**

$pset\text{-}atoms\text{-}branch\ b \equiv \bigcup (atoms\ \text{'}\ set\ b)$

1.2.4 Finiteness of Variables, Subterms, and Subformulae

lemma *finite-vars-term: finite (vars-term t)*

<proof>

lemma *finite-vars-atom*: *finite (vars-atom a)*
<proof>

lemma *finite-vars-fm*: *finite (vars-fm φ)*
<proof>

lemma *finite-vars-branch*: *finite (vars-branch b)*
<proof>

lemma *finite-subterms-term*: *finite (subterms-term l)*
<proof>

lemma *finite-subterms-atom*: *finite (subterms-atom l)*
<proof>

lemma *finite-subterms-fm*: *finite (subterms-fm φ)*
<proof>

lemma *finite-subterms-branch*: *finite (subterms-branch b)*
<proof>

lemma *finite-subfms*: *finite (subfms φ)*
<proof>

lemma *finite-subfms-branch*: *finite (subfms-branch b)*
<proof>

lemma *finite-atoms*: *finite (atoms φ)*
<proof>

lemma *finite-pset-atoms-branch*: *finite (pset-atoms-branch b)*
<proof>

1.2.5 Non-Emptiness of Subterms

lemma *subterms-term-nonempty[simp]*: *subterms-term t \neq {}*
<proof>

lemma *subterms-atom-nonempty[simp]*: *subterms-atom l \neq {}*
<proof>

lemma *subterms-fm-nonempty[simp]*: *subterms-fm $\varphi \neq$ {}*
<proof>

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 ($\langle \vdash - : - \rangle$ [46, 46, 46] 46) **where**

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

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

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

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

$l = Suc\ n \Longrightarrow v \vdash \emptyset n : l$
 $l = v\ x \Longrightarrow v \vdash Var\ x : l$
 $l \neq 0 \Longrightarrow v \vdash t : nat.pred\ l \Longrightarrow v \vdash Single\ t : l$
(*proof*)

definition *type-of-term* :: ('a \Rightarrow nat) \Rightarrow 'a pset-term \Rightarrow nat **where**
type-of-term $v\ t \equiv THE\ l.\ v \vdash t : l$

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*]:

$t1 \in \text{tlvl-terms } (t1 \in_s t2)$
 $t2 \in \text{tlvl-terms } (t2 \in_s t1)$
 $t1 \in \text{tlvl-terms } (t1 =_s t2)$
 $t2 \in \text{tlvl-terms } (t2 =_s t1)$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $P\ b'\ b$
 $\langle proof \rangle$

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

lexpands-fm $b'\ b \implies lexpands\ b'\ b$
 \mid *lexpands-un* $b'\ b \implies lexpands\ b'\ b$
 \mid *lexpands-int* $b'\ b \implies lexpands\ b'\ b$
 \mid *lexpands-diff* $b'\ b \implies lexpands\ b'\ b$
 \mid *lexpands-single* $b'\ b \implies lexpands\ b'\ b$

| *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
 $(\bigwedge s t1 b t2. \text{AF } (s \in_s t1) \in \text{set } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 \sqcap_s t2)] b) \implies$
 $(\bigwedge s t2 b t1. \text{AF } (s \in_s t2) \in \text{set } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 \sqcap_s t2)] b) \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 \sqcap_s t2) \in \text{set } b \implies \text{AT } (s \in_s t1) \in \text{set } b \implies P [\text{AF } (s \in_s t2)] b) \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 \sqcap_s t2) \in \text{set } b \implies \text{AT } (s \in_s t2) \in \text{set } b \implies P [\text{AF } (s \in_s t1)] b) \implies$
 $(\bigwedge s t1 b t2. \text{AT } (s \in_s t1) \in \text{set } b \implies \text{AT } (s \in_s t2) \in \text{set } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AT } (s \in_s t1 \sqcap_s t2)] b) \implies$
 $(\bigwedge s t1 t2 b. \text{AT } (s \in_s t1 -_s t2) \in \text{set } b \implies P [\text{AT } (s \in_s t1), \text{AF } (s \in_s t2)] b) \implies$
 $(\bigwedge s t1 b t2. \text{AF } (s \in_s t1) \in \text{set } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 -_s t2)] b) \implies$
 $(\bigwedge s t2 b t1. \text{AT } (s \in_s t2) \in \text{set } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AF } (s \in_s t1 -_s t2)] b) \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 -_s t2) \in \text{set } b \implies \text{AT } (s \in_s t1) \in \text{set } b \implies P [\text{AT } (s \in_s t2)] b) \implies$
 $(\bigwedge s t1 t2 b. \text{AF } (s \in_s t1 -_s t2) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies P [\text{AF } (s \in_s t1)] b) \implies$
 $(\bigwedge s t1 b t2. \text{AT } (s \in_s t1) \in \text{set } b \implies \text{AF } (s \in_s t2) \in \text{set } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b) \implies P [\text{AT } (s \in_s t1 -_s t2)] b) \implies$
 $(\bigwedge t1 b. \text{Single } t1 \in \text{subterms } (\text{last } b) \implies P [\text{AT } (t1 \in_s \text{Single } t1)] b) \implies$

$$\begin{aligned}
& (\bigwedge s \ t1 \ b. \ AT \ (s \in_s \ Single \ t1) \in \ set \ b \implies P \ [AT \ (s =_s \ t1)] \ b) \implies \\
& (\bigwedge s \ t1 \ b. \ AF \ (s \in_s \ Single \ t1) \in \ set \ b \implies P \ [AF \ (s =_s \ t1)] \ b) \implies \\
& (\bigwedge t1 \ t2 \ b \ l. \ AT \ (t1 =_s \ t2) \in \ set \ b \implies AT \ l \in \ set \ b \implies t1 \in \ tlvl\text{-terms} \ l \implies \\
P \ [AT \ (subst\text{-}tlvl \ t1 \ t2 \ l)] \ b) \implies \\
& (\bigwedge t1 \ t2 \ b \ l. \ AT \ (t1 =_s \ t2) \in \ set \ b \implies AF \ l \in \ set \ b \implies t1 \in \ tlvl\text{-terms} \ l \implies \\
P \ [AF \ (subst\text{-}tlvl \ t1 \ t2 \ l)] \ b) \implies \\
& (\bigwedge t1 \ t2 \ b \ l. \ AT \ (t1 =_s \ t2) \in \ set \ b \implies AT \ l \in \ set \ b \implies t2 \in \ tlvl\text{-terms} \ l \implies \\
P \ [AT \ (subst\text{-}tlvl \ t2 \ t1 \ l)] \ b) \implies \\
& (\bigwedge t1 \ t2 \ b \ l. \ AT \ (t1 =_s \ t2) \in \ set \ b \implies AF \ l \in \ set \ b \implies t2 \in \ tlvl\text{-terms} \ l \implies \\
P \ [AF \ (subst\text{-}tlvl \ t2 \ t1 \ l)] \ b) \implies \\
& (\bigwedge s \ t \ b \ s'. \ AT \ (s \in_s \ t) \in \ set \ b \implies AF \ (s' \in_s \ t) \in \ set \ b \implies P \ [AF \ (s =_s \ s')] \\
b) \implies P \ b' \ b \\
& \langle proof \rangle
\end{aligned}$$

2.2.3 Branching Expansion Rules

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

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

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**

$$\begin{aligned}
& \llbracket AF \ (t1 =_s \ t2) \in \ set \ b; \ t1 \in \ subterms \ (last \ b); \ t2 \in \ subterms \ (last \ b); \\
& \quad \nexists x. \ AT \ (x \in_s \ t1) \in \ set \ b \wedge AF \ (x \in_s \ t2) \in \ set \ b; \\
& \quad \nexists x. \ AT \ (x \in_s \ t2) \in \ set \ b \wedge AF \ (x \in_s \ t1) \in \ set \ b; \\
& \quad x \notin \ vars \ b; \ \neg \ urelem \ (last \ b) \ t1; \ \neg \ urelem \ (last \ b) \ t2 \rrbracket \\
& \implies \text{berpands-wit} \ t1 \ t2 \ x \ \{[AT \ (Var \ x \in_s \ t1), AF \ (Var \ x \in_s \ t2)], \\
& \quad [AT \ (Var \ x \in_s \ t2), AF \ (Var \ x \in_s \ t1)]\} \ b
\end{aligned}$$

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

lemma *berpands-witD*:

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

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

$AF (t1 =_s t2) \in set\ b \ t1 \in subterms\ (last\ b) \ t2 \in subterms\ (last\ b)$
 $\nexists x. AT (x \in_s t1) \in set\ b \wedge AF (x \in_s t2) \in set\ b$
 $\nexists x. AT (x \in_s t2) \in set\ b \wedge AF (x \in_s t1) \in set\ b$
 $\neg urelem\ (last\ b) \ t1 \ \neg urelem\ (last\ b) \ t2$
 $x \notin vars\ b$
 <proof>

inductive *bexpands* :: 'a branch set \Rightarrow 'a branch \Rightarrow bool **where**
bexpands-nowit $bs' \ b \Longrightarrow bexpands\ bs' \ b$
 | *bexpands-wit* $t1 \ t2 \ x \ bs' \ b \Longrightarrow bexpands\ bs' \ b$

lemma *bexpands-disjnt*:
assumes *bexpands* $bs' \ b \ b' \in bs'$
shows $set\ b \cap set\ b' = \{\}$
 <proof>

lemma *bexpands-branch-not-Nil*:
assumes *bexpands* $bs' \ b \ b' \in bs'$
shows $b' \neq []$
 <proof>

lemma *bexpands-nonempty*: *bexpands* $bs' \ b \Longrightarrow bs' \neq \{\}$
 <proof>

lemma *bexpands-strict-mono*:
assumes *bexpands* $bs' \ b \ b' \in bs'$
shows $set\ b \subset set\ (b' @ b)$
 <proof>

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 $b \ b$
 | *lexpandss* $b3 \ b2 \Longrightarrow set\ b2 \subset set\ (b3 @ b2) \Longrightarrow expandss\ b2 \ b1 \Longrightarrow expandss\ (b3 @ b2) \ b1$
 | *bexpandss* $bs \ b2 \Longrightarrow b3 \in bs \Longrightarrow expandss\ b2 \ b1 \Longrightarrow expandss\ (b3 @ b2) \ b1$

lemma *expandss-trans*: *expandss* $b3 \ b2 \Longrightarrow expandss\ b2 \ b1 \Longrightarrow expandss\ b3 \ b1$
 <proof>

lemma *expandss-suffix*:
expandss $b1 \ b2 \Longrightarrow suffix\ b2 \ b1$
 <proof>

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

lemma *expandss-last-eq*[simp]:
 $expandss\ b'\ b \implies b \neq [] \implies last\ b' = last\ b$
(proof)

lemma *expandss-not-Nil*:
 $expandss\ b'\ b \implies b \neq [] \implies b' \neq []$
(proof)

2.2.5 Well-Formed Branch

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

lemma *wf-branch-singleton*[simp]: *wf-branch* $[\varphi]$
(proof)

lemma *wf-branch-not-Nil*[simp, intro?]: *wf-branch* $b \implies b \neq []$
(proof)

lemma *wf-branch-expandss*: *wf-branch* $b \implies expandss\ b'\ b \implies wf-branch\ b'$
(proof)

lemma *wf-branch-lexpands*:
 $wf-branch\ b \implies lexpands\ b'\ b \implies set\ b \subset set\ (b' @ b) \implies wf-branch\ (b' @ b)$
(proof)

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)  
  <proof>
```

```
lemma finite-set-suc-atom[simp]: finite (set-suc-atom a)  
  <proof>
```

```
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]*: $\neg is-Succ-normal-term (Succ 0 t)$
 $\langle proof \rangle$

lemma *is-Succ-normal-term-SuccD[simp]*: $is-Succ-normal-term (Succ n t) \implies is-Succ-normal-term t$
 $\langle proof \rangle$

fun *is-Succ-normal-atom* :: $'a \text{ suc-atom} \Rightarrow bool$ **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* :: $('a \Rightarrow nat) \Rightarrow 'a \text{ suc-term} \Rightarrow nat$ **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* :: $('a \Rightarrow nat) \Rightarrow 'a \text{ suc-atom} \Rightarrow bool$ **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* :: $('a \Rightarrow 'a \text{ suc-term}) \Rightarrow 'a \text{ suc-term} \Rightarrow 'a \text{ suc-term}$ **where**
 $subst-term \sigma (Var x) = succ 0 (\sigma x)$
 $| subst-term - Zero = Zero$
 $| subst-term \sigma (Succ n t) = succ n (subst-term \sigma t)$

fun *subst-atom* :: $('a \Rightarrow 'a \text{ suc-term}) \Rightarrow 'a \text{ suc-atom} \Rightarrow 'a \text{ suc-atom}$ **where**
 $subst-atom \sigma (Eq t1 t2) = Eq (subst-term \sigma t1) (subst-term \sigma t2)$
 $| subst-atom \sigma (NEq t1 t2) = NEq (subst-term \sigma t1) (subst-term \sigma t2)$

lemma *I-term-succ*: $I-term v (succ n t) = I-term v (Succ n t)$
 $\langle proof \rangle$

lemma *is-Succ-normal-succ[simp]*: $is-Succ-normal (succ n t)$
 $\langle proof \rangle$

lemma *is-Succ-normal-subst-term[simp]*: $is-Succ-normal (subst-term \sigma t)$
 $\langle proof \rangle$

lemma *is-Succ-normal-subst-atom*[simp]: *is-Succ-normal* (subst-atom σ a)
 ⟨proof⟩

lemma *is-NEq-subst-atom*[simp]:
is-NEq (subst-atom v a) \longleftrightarrow *is-NEq* a
 ⟨proof⟩

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)
 ⟨proof⟩

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
 ⟨proof⟩

termination ⟨proof⟩

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
 ⟨proof⟩

lemma *I-term-subst-term*:
assumes *I-atom* v (Eq (Var x) t)

shows $I\text{-term } v \text{ (subst-term (Var(x := t)) s) = I-term } v \text{ s}$
<proof>

lemma *I-atom-subst-atom:*
assumes $I\text{-atom } v \text{ (Eq (Var x) t)}$
shows $I\text{-atom } v \text{ (subst-atom (Var(x := t)) a) } \longleftrightarrow I\text{-atom } v \text{ a}$
<proof>

lemma *I-atom-solve-None:*
assumes $\text{solve } es = \text{None } \forall a \in \text{set } es. \text{is-normal } a$
shows $\exists a \in \text{set } es. \neg I\text{-atom } v \text{ a}$
<proof>

lemma *set-suc-term-succ[simp]:* $\text{set-suc-term (succ n t) = set-suc-term } t$
<proof>

lemma *not-mem-subst-term-self:*
assumes $x \notin \text{set-suc-term } t$
shows $x \notin \text{set-suc-term (subst-term (Var(x := t)) s)}$
<proof>

lemma *not-mem-subst-atom-self:*
assumes $x \notin \text{set-suc-term } t$
shows $x \notin \text{set-suc-atom (subst-atom (Var(x := t)) a)}$
<proof>

lemma *not-mem-subst-term:*
assumes $z \notin \text{set-suc-term } t \ z \notin \text{set-suc-term } s$
shows $z \notin \text{set-suc-term (subst-term (Var(x := t)) s)}$
<proof>

lemma *not-mem-subst-atom:*
assumes $z \notin \text{set-suc-term } t \ z \notin \text{set-suc-atom } a$
shows $z \notin \text{set-suc-atom (subst-atom (Var(x := t)) a)}$
<proof>

lemma *not-mem-suc-atom-solve:*
assumes $\text{solve } es = \text{Some } ss \ \forall a \in \text{set } es. \text{is-normal } a$
assumes $\forall a \in \text{set } es. z \notin \text{set-suc-atom } a$
shows $\forall a \in \text{set } ss. z \notin \text{set-suc-atom } a$
<proof>

lemma *not-mem-suc-atom-if-solve:*
assumes $\text{solve } es = \text{Some (Eq (Var x) t \# ss) } \forall a \in \text{set } es. \text{is-normal } a$
shows $\forall a \in \text{set } ss. x \notin \text{set-suc-atom } a$
<proof>

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

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

lemma *assign-succ*:

$assign (Eq (Var x) (succ\ n\ t) \# ss) = assign (Eq (Var x) (Succ\ n\ t) \# ss)$
 <proof>

lemma *I-term-fun-upd*:

assumes $x \notin set\ suc\ term\ t$
shows $I\ term\ (v(x := s))\ t = I\ term\ v\ t$
 <proof>

lemma *I-atom-fun-upd*:

assumes $x \notin set\ suc\ atom\ a$
shows $I\ atom\ (v(x := s))\ a = I\ atom\ v\ a$
 <proof>

lemma *I-atom-fun-updI*:

assumes $x \notin set\ suc\ atom\ a$ $I\ atom\ v\ a$
shows $I\ atom\ (v(x := s))\ a$
 <proof>

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

assumes $solve\ es = Some\ ss\ \forall a \in set\ es.\ is\ normal\ a$
shows $\forall a \in set\ ss.\ I\ atom\ (assign\ ss)\ a$
 <proof>

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

assumes $solve\ es = Some\ ss\ \forall a \in set\ es.\ is\ normal\ a$
shows $(\forall a \in set\ ss.\ I\ atom\ v\ a) \longleftrightarrow (\forall a \in set\ es.\ I\ atom\ v\ a)$
 <proof>

lemma *assign-minimal-if-solve-Some*:

assumes $solve\ es = Some\ ss\ \forall a \in set\ es.\ is\ normal\ a$
assumes $\forall a \in set\ ss.\ I\ atom\ v\ a$
shows $assign\ ss\ z \leq v\ z$
 <proof>

fun *elim-NEq-Zero-aux* :: ('a::fresh) set \Rightarrow 'a suc-atom list \Rightarrow 'a suc-atom list
where

$elim\ NEq\ Zero\ aux\ -\ [] = []$
 | $elim\ NEq\ Zero\ aux\ us\ (NEq\ (Var\ x)\ Zero\ \#\ es) =$
 $(let\ fx = fresh\ us\ x\ in\ Eq\ (Var\ x)\ (Succ\ 1\ (Var\ fx))) \#\ elim\ NEq\ Zero\ aux\ (insert$

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

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

lemma $\text{is-normal-elim-NEq-Zero-aux}$:
assumes $\forall a \in \text{set } es.\ \text{is-Eq } a \longrightarrow \text{is-normal } a$
assumes $\forall a \in \text{set } es.\ \text{is-NEq } a \longrightarrow (\exists x.\ a = \text{NEq } (\text{Var } x)\ \text{Zero})$
shows $\forall a \in \text{set } (\text{elim-NEq-Zero-aux } us\ es).\ \text{is-normal } a$
 $\langle \text{proof} \rangle$

lemma $\text{is-normal-elim-NEq-Zero}$:
assumes $\forall a \in \text{set } es.\ \text{is-Eq } a \longrightarrow \text{is-normal } a$
assumes $\forall a \in \text{set } es.\ \text{is-NEq } a \longrightarrow (\exists x.\ a = \text{NEq } (\text{Var } x)\ \text{Zero})$
shows $\forall a \in \text{set } (\text{elim-NEq-Zero } es).\ \text{is-normal } a$
 $\langle \text{proof} \rangle$

lemma $\text{I-atom-Var-NEq-Zero-if-I-atom-Var-Eq-Succ}$:
 $\text{I-atom } v\ (\text{Eq } (\text{Var } x)\ (\text{Succ } 1\ (\text{Var } fx))) \Longrightarrow \text{I-atom } v\ (\text{NEq } (\text{Var } x)\ \text{Zero})$
 $\langle \text{proof} \rangle$

lemma $\text{I-atom-if-I-atom-elim-NEq-Zero-aux}$:
assumes $\forall a \in \text{set } (\text{elim-NEq-Zero-aux } us\ es).\ \text{I-atom } v\ a$
shows $\forall a \in \text{set } es.\ \text{I-atom } v\ a$
 $\langle \text{proof} \rangle$

lemma $\text{I-atom-if-I-atom-elim-NEq-Zero}$:
assumes $\forall a \in \text{set } (\text{elim-NEq-Zero } es).\ \text{I-atom } v\ a$
shows $\forall a \in \text{set } es.\ \text{I-atom } v\ a$
 $\langle \text{proof} \rangle$

lemma $\text{I-term-if-eq-on-set-suc-term}$:
assumes $\forall x \in \text{set-suc-term } t.\ v'\ x = v\ x$
shows $\text{I-term } v'\ t = \text{I-term } v\ t$
 $\langle \text{proof} \rangle$

lemma $\text{I-atom-if-eq-on-set-suc-atom}$:
assumes $\forall x \in \text{set-suc-atom } a.\ v'\ x = v\ x$
shows $\text{I-atom } v'\ a = \text{I-atom } v\ a$
 $\langle \text{proof} \rangle$

lemma $\text{not-mem-set-suc-atom-elim-NEq-zero-aux}$:
assumes $\text{finite } us\ \bigcup (\text{set-suc-atom } ' \text{ set } es) \subseteq us$
assumes $a \in \text{set } (\text{elim-NEq-Zero-aux } us\ es)$
assumes $x \in us - \bigcup (\text{set-suc-atom } ' \text{ set } es)$
shows $x \notin \text{set-suc-atom } a$
 $\langle \text{proof} \rangle$

lemma *I-atom-elim-NEq-Zero-aux-if-I-atom:*
assumes $\bigcup(\text{set-suc-atom } ' \text{ set es}) \subseteq \text{us finite us}$
assumes $\forall a \in \text{set es. I-atom } v a$
obtains v' **where** $\forall a \in \text{set } (\text{elim-NEq-Zero-aux us es}). \text{I-atom } v' a$
 $\forall x \in \text{us. } v' x = v x$
 $\langle \text{proof} \rangle$

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

lemma *not-I-atom-if-solve-elim-NEq-Zero-None:*
assumes $\forall a \in \text{set es. is-Eq } a \longrightarrow \text{is-normal } a$
assumes $\forall a \in \text{set es. is-NEq } a \longrightarrow (\exists x. a = \text{NEq } (\text{Var } x) \text{ Zero})$
assumes $\text{solve } (\text{elim-NEq-Zero es}) = \text{None}$
shows $\exists a \in \text{set es. } \neg \text{I-atom } v a$
 $\langle \text{proof} \rangle$

lemma
assumes $\forall a \in \text{set es. is-Eq } a \longrightarrow \text{is-normal } a$
assumes $\forall a \in \text{set es. is-NEq } a \longrightarrow (\exists x. a = \text{NEq } (\text{Var } x) \text{ Zero})$
assumes $\text{solve } (\text{elim-NEq-Zero es}) = \text{Some } ss$
shows *I-atom-assign-if-solve-elim-NEq-Zero-Some:*
 $\forall a \in \text{set es. I-atom } (\text{assign } ss) a$
and *I-atom-assign-minimal-if-solve-elim-NEq-Zero-Some:*
 $\llbracket \forall a \in \text{set es. I-atom } v a; z \in \bigcup(\text{set-suc-atom } ' \text{ set es}) \rrbracket$
 $\implies \text{assign } ss z \leq v z$
 $\langle \text{proof} \rangle$

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 $v \vdash t : l1 \implies v \vdash t : l2 \implies l2 = l1$
 $\langle \text{proof} \rangle$

lemma *type-of-term-if-types-term:*
shows $v \vdash t : l \implies \text{type-of-term } v t = l$
 $\langle \text{proof} \rangle$

lemma *types-term-if-mem-subterms-term:*
assumes $s \in \text{subterms } t$

assumes $v \vdash t : lt$
shows $\exists ls. v \vdash s : ls$
 $\langle proof \rangle$

lemma *is-Var-if-types-term-0*:
shows $v \vdash t : 0 \implies is-Var\ t$
 $\langle proof \rangle$

lemma *is-Var-if-urelem'*: $urelem'\ v\ \varphi\ t \implies is-Var\ t$
 $\langle proof \rangle$

lemma *is-Var-if-urelem*: $urelem\ \varphi\ t \implies is-Var\ t$
 $\langle proof \rangle$

lemma *types-fmD*:
 $v \vdash And\ p\ q \implies v \vdash p$
 $v \vdash And\ p\ q \implies v \vdash q$
 $v \vdash Or\ p\ q \implies v \vdash p$
 $v \vdash Or\ p\ q \implies v \vdash q$
 $v \vdash Neg\ p \implies v \vdash p$
 $v \vdash Atom\ a \implies v \vdash a$
 $\langle proof \rangle$

lemma *types-fmI*:
 $v \vdash p \implies v \vdash q \implies v \vdash And\ p\ q$
 $v \vdash p \implies v \vdash q \implies v \vdash Or\ p\ q$
 $v \vdash p \implies v \vdash Neg\ p$
 $v \vdash a \implies v \vdash Atom\ a$
 $\langle proof \rangle$

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 $v \vdash s \in_s t1\ v \vdash s \in_s t2$
 $\langle proof \rangle$

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 $\varphi :: 'a\ pset-fm$
assumes $v \vdash \varphi$
assumes $f\ t1\ t2 \in subterms\ \varphi\ f \in \{(\sqcup_s), (\sqcap_s), (-_s)\}$
obtains lt **where** $v \vdash t1 : lt\ v \vdash t2 : lt$

<proof>

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>

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>

lemma *types-sym-Equal:*

assumes $v \vdash t1 =_s t2$

shows $v \vdash t2 =_s t1$

<proof>

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 $v \vdash \varphi$

<proof>

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 $v \vdash \varphi$

<proof>

lemma *types-term-if-on-vars-eq:*

assumes $\forall x \in \text{vars } t. v' \ x = v \ x$

shows $v' \vdash t : l \longleftrightarrow v \vdash t : l$

<proof>

lemma *types-pset-atom-if-on-vars-eq:*

fixes $a :: 'a \text{ pset-atom}$

assumes $\forall x \in \text{vars } a. v' \ x = v \ x$

shows $v' \vdash a \longleftrightarrow v \vdash a$

<proof>

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 $v' \vdash \varphi \longleftrightarrow v \vdash \varphi$
<proof>

lemma *types-term-fun-upd*:
assumes $x \notin \text{vars } t$
shows $v(x := l) \vdash t : l \longleftrightarrow v \vdash t : l$
<proof>

lemma *types-pset-atom-fun-upd*:
fixes $a :: 'a \text{ pset-atom}$
assumes $x \notin \text{vars } a$
shows $v(x := l) \vdash a \longleftrightarrow v \vdash a$
<proof>

lemma *types-pset-fm-fun-upd*:
fixes $\varphi :: 'a \text{ pset-fm}$
assumes $x \notin \text{vars } \varphi$
shows $v(x := l) \vdash \varphi \longleftrightarrow v \vdash \varphi$
<proof>

lemma *types-bexpands-wit*:
fixes $b :: 'a \text{ branch}$ **and** $bs' :: 'a \text{ branch set}$
assumes $\text{bexpands-wit } t1 \ t2 \ x \ bs' \ b \ b \neq []$
assumes $\bigwedge(\varphi :: 'a \text{ pset-fm}). \varphi \in \text{set } b \implies v \vdash \varphi$
obtains l **where** $\forall \varphi \in \text{set } b. v(x := l) \vdash \varphi$
 $\forall b' \in bs'. \forall \varphi \in \text{set } b'. v(x := l) \vdash \varphi$
<proof>

lemma *types-expandss*:
fixes $b \ b' :: 'a \text{ branch}$
assumes $\text{expandss } b' \ b \ b \neq []$
assumes $\bigwedge \varphi. \varphi \in \text{set } b \implies v \vdash \varphi$
obtains v' **where** $\forall x \in \text{vars } b. v' \ x = v \ x \ \forall \varphi \in \text{set } b'. v' \vdash \varphi$
<proof>

lemma *urelem-invar-if-wf-branch*:
assumes $\text{wf-branch } b$
assumes $\text{urelem } (\text{last } b) \ x \ x \in \text{subterms } (\text{last } b)$
shows $\exists v. \forall \varphi \in \text{set } b. \text{urelem}' \ v \ \varphi \ x$
<proof>

lemma *not-types-term-0-if-types-term*:
fixes $s :: 'a \text{ pset-term}$
assumes $f \ t1 \ t2 \in \text{subterms } s \ f \in \{(\sqcap_s), (\sqcup_s), (-_s)\}$
assumes $v \vdash f \ t1 \ t2 : l$
shows $\neg v \vdash t1 : 0 \ \neg v \vdash t2 : 0$
<proof>

lemma *types-term-subterms*:

assumes $t \in \text{subterms } s$
assumes $v \vdash s : ls$
obtains lt **where** $v \vdash t : lt$
 $\langle \text{proof} \rangle$

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

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** $v \vdash t : lt$
 $\langle \text{proof} \rangle$

lemma *not-urelem-comps-if-compound*:
assumes $f t1 t2 \in \text{subterms } \varphi \ f \in \{(\Pi_s), (\sqcup_s), (-_s)\}$
shows $\neg \text{urelem } \varphi \ t1 \ \neg \text{urelem } \varphi \ t2$
 $\langle \text{proof} \rangle$

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

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

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*:

$\forall a \in set (constrs-term n t). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
 $\langle proof \rangle$

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

$\forall a \in set (constrs-atom n a). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
 $\langle proof \rangle$

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

$\forall a \in set (constrs-fm n \varphi). MLSS-Suc-Theory.is-Eq a \longrightarrow is-Succ-normal a$
 $\langle proof \rangle$

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

$\forall a \in set (constrs-term n t). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq$
 $(SVar x) Zero)$
 $\langle proof \rangle$

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

$\forall a \in set (constrs-atom n a). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq$
 $(SVar x) Zero)$
 $\langle proof \rangle$

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

$\forall a \in set (constrs-fm n \varphi). MLSS-Suc-Theory.is-NEq a \longrightarrow (\exists x. a = SNEq (SVar$
 $x) Zero)$
 $\langle proof \rangle$

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 $(\lambda x. v \ (n \ (\text{Var } x))) \vdash t : v \ (n \ t)$
 $\langle \text{proof} \rangle$

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 $(\lambda x. v \ (n \ (\text{Var } x))) \vdash a$
 $\langle \text{proof} \rangle$

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 $(\lambda x. v \ (n \ (\text{Var } x))) \vdash \varphi$
 $\langle \text{proof} \rangle$

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

includes *no member-ASCII-syntax*
assumes $\text{inj-on } n \ T \ \text{subterms } t \subseteq T$
assumes $v \vdash t : k$
shows $(\forall e \in \text{set } (\text{constrs-term } n \ t)).$
 $\text{MLSS-Suc-Theory.I-atom } (\lambda x. \text{type-of-term } v \ (\text{inv-into } T \ n \ x)) \ e)$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *inv-into-f-eq-if-subs:*

assumes $\text{inj-on } f \ B \ A \subseteq B \ y \in f \ 'A$
shows $\text{inv-into } B \ f \ y = \text{inv-into } A \ f \ y$
 $\langle \text{proof} \rangle$

lemma *UN-set-suc-atom-constrs-term-eq-image-subterms:*

$\bigcup (\text{set-suc-atom } ' \ \text{set } (\text{constrs-term } n \ t)) = n \ ' \ \text{subterms } t$
 $\langle \text{proof} \rangle$

lemma *UN-set-suc-atom-constrs-atom-eq-image-subterms:*
 $\bigcup (\text{set-suc-atom } ' \text{ set } (\text{constrs-atom } n \ a)) = n \ ' \ \text{subterms } a$
 $\langle \text{proof} \rangle$

lemma *UN-set-suc-atom-constrs-fm-eq-image-subterms:*
 $\bigcup (\text{set-suc-atom } ' \ \text{set } (\text{constrs-fm } n \ \varphi)) = n \ ' \ \text{subterms } \varphi$
 $\langle \text{proof} \rangle$

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

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 $k' \leq k$
 $\langle \text{proof} \rangle$

lemma *constrs-term-subs-constrs-term:*
assumes $s \in \text{subterms } t$
shows $\text{set } (\text{constrs-term } n \ s) \subseteq \text{set } (\text{constrs-term } n \ t)$
 $\langle \text{proof} \rangle$

lemma *constrs-term-subs-constrs-atom:*
assumes $t \in \text{subterms } a$
shows $\text{set } (\text{constrs-term } n \ t) \subseteq \text{set } (\text{constrs-atom } n \ a)$
 $\langle \text{proof} \rangle$

lemma *constrs-term-subs-constrs-fm:*
assumes $t \in \text{subterms } \varphi$
shows $\text{set } (\text{constrs-term } n \ t) \subseteq \text{set } (\text{constrs-fm } n \ \varphi)$
 $\langle \text{proof} \rangle$

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

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>
```

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
<proof>

lemma (in *fin-digraph*) *finite-parents[intro]*: *finite* (*parents* G s)
<proof>

lemma (in *fin-digraph*) *finite-ancestors[intro]*: *finite* (*ancestors* G s)
<proof>

lemma (in *wf-digraph*) *in-ancestors-if-dominates[simp, intro]*:
assumes $s \rightarrow_G t$
shows $s \in$ *ancestors* G t
<proof>

lemma (in *wf-digraph*) *ancestors-mono*:
assumes $s \in$ *ancestors* G t
shows *ancestors* G $s \subseteq$ *ancestors* G t
<proof>

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

lemma *ancestors-asym*:

assumes $s \in \text{ancestors } G \ t$
shows $t \notin \text{ancestors } G \ s$
 $\langle \text{proof} \rangle$

lemma *ancestors-strict-mono*:
assumes $s \in \text{ancestors } G \ t$
shows $\text{ancestors } G \ s \subset \text{ancestors } G \ t$
 $\langle \text{proof} \rangle$

lemma *card-ancestors-strict-mono*:
assumes $s \rightarrow_G \ t$
shows $\text{card} (\text{ancestors } G \ s) < \text{card} (\text{ancestors } G \ t)$
 $\langle \text{proof} \rangle$

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*: $B \cap T = \{\}$ *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*: $\text{finite } (B \cup T)$
 $\langle \text{proof} \rangle$

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

function *realise* $:: 'a \Rightarrow \text{hf}$ **where**
 $x \in B \Longrightarrow \text{realise } x = \text{HF} (\text{realise } ' \text{parents } G \ x) \sqcup \text{HF} (I \ ' \ \text{eq-class } x)$
 $| t \in T \Longrightarrow \text{realise } t = \text{HF} (\text{realise } ' \ \text{parents } G \ t)$
 $| x \notin B \cup T \Longrightarrow \text{realise } x = 0$
 $\langle \text{proof} \rangle$

termination
 $\langle \text{proof} \rangle$

lemma *finite-realisation-parents[simp, intro!]*: $\text{finite} (\text{realise } ' \ \text{parents } G \ t)$
 $\langle \text{proof} \rangle$

function *height* $:: 'a \Rightarrow \text{nat}$ **where**
 $\forall s. \neg s \rightarrow_G t \Longrightarrow \text{height } t = 0$

| $s \rightarrow_G t \implies \text{height } t = \text{Max} (\text{height } \text{' parents } G t) + 1$
<proof>

termination

<proof>

lemma *height-cases'*:

obtains

(Zero) $\text{height } t = 0$

| (Suc-Max) s **where** $s \rightarrow_G t$ $\text{height } t = \text{height } s + 1$

<proof>

lemma *lemma1-1*:

assumes $s \rightarrow_G t$

shows $\text{height } s < \text{height } t$

<proof>

lemma *dominates-if-mem-realisation*:

assumes $\bigwedge x y. I x \neq \text{realise } y$

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

obtains s' **where** $s' \rightarrow_G t$ $\text{realise } s = \text{realise } s'$

<proof>

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 $\text{Max } A \leq \text{Max } B$

<proof>

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 $\text{height } t1 \leq \text{height } t2$

<proof>

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 $\text{height } t1 = \text{height } t2$

<proof>

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>

end

```

theory MLSS-HF-Extras
  imports HereditarilyFinite.Rank
begin

lemma hcard-ord-of[simp]:
  hcard (ord-of n) = n
  ⟨proof⟩

lemma hcard-HF: finite A  $\implies$  hcard (HF A) = card A
  ⟨proof⟩

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 $x \in \text{set } b$
 ⟨proof⟩

lemma *not-lin-satD*: $\neg \text{lin-sat } b \implies \exists b'. \text{lexpands } b' b \wedge \text{set } b \subset \text{set } (b' @ b)$
 ⟨proof⟩

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

lemma *satD*:
assumes *sat* b
shows *lin-sat* $b \nexists bs'. \text{berpands } bs' b$
 ⟨proof⟩

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 $b \equiv \{c \in \text{wits } b. \forall t \in \text{subterms } (\text{last } b). \\ AT (\text{Var } c =_s t) \notin \text{set } b \wedge AT (t =_s \text{Var } c) \notin \text{set } b\}$

lemma *wits-singleton[simp]*: $wits\ [\varphi] = \{\}$
<proof>

lemma *pwits-singleton[simp]*: $pwits\ [\varphi] = \{\}$
<proof>

lemma *pwitsD*:

assumes $c \in pwits\ b$

shows $c \in wits\ b$

$t \in subterms\ (last\ b) \implies AT\ (Var\ c =_s\ t) \notin set\ b$

$t \in subterms\ (last\ b) \implies AT\ (t =_s\ Var\ c) \notin set\ b$

<proof>

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 $c \in pwits\ b$

<proof>

lemma *finite-wits*: $finite\ (wits\ b)$

<proof>

lemma *finite-pwits*: $finite\ (pwits\ b)$

<proof>

lemma *lexpands-subterms-branch-eq*:

$lexpands\ b'\ b \implies b \neq [] \implies subterms\ (b' @ b) = subterms\ b$

<proof>

lemma *lexpands-vars-branch-eq*:

$lexpands\ b'\ b \implies b \neq [] \implies vars\ (b' @ b) = vars\ b$

<proof>

lemma *bexpands-nowit-subterms-branch-eq*:

$bexpands-nowit\ bs'\ b \implies b' \in bs' \implies b \neq [] \implies subterms\ (b' @ b) = subterms\ b$

<proof>

lemma *bexpands-nowit-vars-branch-eq*:

$bexpands-nowit\ bs'\ b \implies b' \in bs' \implies b \neq [] \implies vars\ (b' @ b) = vars\ b$

<proof>

lemma *lexpands-wits-eq*:

$lexpands\ b'\ b \implies b \neq [] \implies wits\ (b' @ b) = wits\ b$

<proof>

lemma *bexpands-nowit-wits-eq*:

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

shows $wits (b' @ b) = wits b$
 ⟨proof⟩

lemma *bexpands-wit-vars-branch-eq*:
assumes *bexpands-wit* $t1 t2 x bs' b b' \in bs' b \neq []$
shows $vars (b' @ b) = insert x (vars b)$
 ⟨proof⟩

lemma *bexpands-wit-wits-eq*:
assumes *bexpands-wit* $t1 t2 x bs' b b' \in bs' b \neq []$
shows $wits (b' @ b) = insert x (wits b)$
 ⟨proof⟩

lemma *lexpands-pwits-subs*:
assumes *lexpands* $b' b b \neq []$
shows $pwits (b' @ b) \subseteq pwits b$
 ⟨proof⟩

no-new-subterms

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

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

lemma *Var-mem-subterms-branch-and-not-in-wits*:
assumes $Var v \in subterms b \ v \notin wits b$
shows $v \in vars (last b)$
 ⟨proof⟩

lemma *subterms-branch-cases*:
assumes $t \in subterms b \ t \notin Var \text{ ' } wits b$
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⟩

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 $\bigwedge t. \text{Single } t \in \text{subterms } b \implies \text{Single } t \in \text{subterms } (\text{last } b)$
shows *no-new-subterms* b
<proof>

lemma *no-new-subtermsD*:

assumes *no-new-subterms* b
shows $\bigwedge n. \emptyset n \in \text{subterms } b \implies \emptyset n \in \text{subterms } (\text{last } b)$
 $\bigwedge t1\ t2. t1 \sqcup_s t2 \in \text{subterms } b \implies t1 \sqcup_s t2 \in \text{subterms } (\text{last } b)$
 $\bigwedge t1\ t2. t1 \sqcap_s t2 \in \text{subterms } b \implies t1 \sqcap_s t2 \in \text{subterms } (\text{last } b)$
 $\bigwedge t1\ t2. t1 -_s t2 \in \text{subterms } b \implies t1 -_s t2 \in \text{subterms } (\text{last } b)$
 $\bigwedge t. \text{Single } t \in \text{subterms } b \implies \text{Single } t \in \text{subterms } (\text{last } b)$
<proof>

lemma *lexpands-no-new-subterms*:

assumes *lexpands* $b' b b \neq []$
assumes *no-new-subterms* b
shows *no-new-subterms* $(b' @ b)$
<proof>

lemma *subterms-branch-subterms-atomI*:

assumes *Atom* $l \in \text{set } b\ t \in \text{subterms-atom } l$
shows $t \in \text{subterms-branch } b$
<proof>

lemma *bexpands-nowit-no-new-subterms*:

assumes *bexpands-nowit* $bs' b b' \in bs' b b \neq []$
assumes *no-new-subterms* b
shows *no-new-subterms* $(b' @ b)$
<proof>

lemma *bexpands-wit-no-new-subterms*:

assumes *bexpands-wit* $t1\ t2\ x\ bs' b b \neq []\ b' \in bs'$
assumes *no-new-subterms* b
shows *no-new-subterms* $(b' @ b)$
<proof>

lemma *bexpands-no-new-subterms*:

assumes *bexpands* $bs' b b \neq []\ b' \in bs'$
assumes *no-new-subterms* b
shows *no-new-subterms* $(b' @ b)$
<proof>

lemma *expandss-no-new-subterms*:

assumes *expandss* $b' b b \neq []\ \text{no-new-subterms } b$
shows *no-new-subterms* b'
<proof>

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

<proof>

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

<proof>

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)

<proof>

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)

<proof>

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)

<proof>

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)

<proof>

lemma *expands-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 *expands* $b' b b \neq []$

assumes $P b$

shows $P b'$

<proof>

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 $\text{pwits } (b' @ b) = \text{pwits } b$

<proof>

lemma *lexpands-un-pwits-eq*:

assumes *lexpands-un* $b' b b \neq []$

assumes $\forall c \in \text{pwits } b. \forall t \in \text{wits-subterms } b.$

$AT (Var c =_s t) \notin \text{set } b \wedge AT (t =_s Var c) \notin \text{set } b \wedge AT (t \in_s Var c) \notin \text{set } b$

shows $\text{pwits } (b' @ b) = \text{pwits } b$

<proof>

lemma *lexpands-int-pwits-eq*:

assumes *lexpands-int* $b' b b \neq []$

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>

lemma *lexpands-diff-pwits-eq*:

assumes *lexpands-diff* $b' b b \neq []$

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>

lemma *bexpands-nowit-pwits-eq*:

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

assumes $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$

shows $\text{pwits } (b' @ b) = \text{pwits } b$

<proof>

lemma *bexpands-wit-pwits-eq*:

assumes *bexpands-wit* $t1 t2 x bs' b b' \in bs' b \neq []$

shows $\text{pwits } (b' @ b) = \text{insert } x (\text{pwits } b)$

<proof>

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 $\forall c \in \text{pwits } (b' @ b). \forall t \in \text{wits-subterms } (b' @ b). P \ (b' @ b) \ c \ t$
 $\langle \text{proof} \rangle$

lemma *lemma-2-beexpands*:

defines $P \equiv (\lambda b \ c \ t. AT \ (Var \ c =_s \ t) \notin \text{set } b \wedge AT \ (t =_s \ Var \ c) \notin \text{set } b$
 $\quad \wedge AT \ (t \in_s \ Var \ c) \notin \text{set } b)$
assumes *beexpands* $bs' \ b \ b' \in bs' \ b \neq []$
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 $\forall c \in \text{pwits } (b' @ b). \forall t \in \text{wits-subterms } (b' @ b). P \ (b' @ b) \ c \ t$
 $\langle \text{proof} \rangle$

lemma *subterms-branch-eq-if-wf-branch*:

assumes *wf-branch* b
shows *subterms-branch* $b = \text{wits-subterms } b$
 $\langle \text{proof} \rangle$

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*:
 $\forall \varphi \in \text{set } b. \neg \text{is-literal } \varphi \longrightarrow \text{vars } \varphi \cap \text{wits } b = \{\}$
 $\langle \text{proof} \rangle$

lemma *lemma-2*:

assumes *wf-branch* b
assumes $c \in \text{pwits } b$
shows $AT \ (Var \ c =_s \ t) \notin \text{set } b \wedge AT \ (t =_s \ Var \ c) \notin \text{set } b \wedge AT \ (t \in_s \ Var \ c) \notin \text{set } b$
 $\langle \text{proof} \rangle$

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)

$\langle \text{proof} \rangle$

lemma *urelems-subsubterms*: *urelems* $b \subseteq \text{subterms } b$

$\langle \text{proof} \rangle$

lemma *is-Var-if-mem-urelems*: $t \in \text{urelems } b \implies \text{is-Var } t$

$\langle \text{proof} \rangle$

lemma *urelems-subvars*: *urelems* $b \subseteq \text{Var } \text{'vars } b$

$\langle \text{proof} \rangle$

lemma *types-term-inf*:

includes *no member-ASCII-syntax*

assumes $v1 \vdash t : l1$ $v2 \vdash t : l2$

shows $\text{inf } v1 \ v2 \vdash t : \text{inf } l1 \ l2$

<proof>

lemma *types-pset-atom-inf*:

fixes $a :: 'a \text{ pset-atom}$

assumes $v1 \vdash a$ $v2 \vdash a$

shows $\text{inf } v1 \ v2 \vdash a$

<proof>

lemma *types-pset-fm-inf*:

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

assumes $v1 \vdash \varphi$ $v2 \vdash \varphi$

shows $\text{inf } v1 \ v2 \vdash \varphi$

<proof>

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>

lemma *mem-urelems-if-urelem-last*:

assumes *wf-branch* b

assumes *urelem* ($\text{last } b$) $x \in \text{subterms } (\text{last } b)$

shows $x \in \text{urelems } b$

<proof>

lemma *not-urelem-comps-if-compound*:

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

shows $t1 \notin \text{urelems } b$ $t2 \notin \text{urelems } b$

<proof>

Realization of an Open Branch

definition *base-vars* $b \equiv \text{Var } ' \text{pwits } b \cup \text{urelems } b$

lemma *finite-base-vars*: *finite* (*base-vars* b)

<proof>

lemma *pwits-sub-base-vars*:

shows $\text{Var } ' \text{pwits } b \subseteq \text{base-vars } b$

<proof>

lemma *base-vars-sub-vars*: *base-vars* $b \subseteq \text{Var } ' \text{vars } b$

<proof>

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 = $\{(s, t). AT (s \in_s t) \in \text{set } b\}$,
 tail = *fst*,
 head = *snd*)

lemma *base-vars-Un-subterms'-eq-subterms*:
base-vars b \cup *subterms'* b = *subterms* b
<proof>

lemma *finite-base-vars-Un-subterms'*: *finite* (*base-vars* b \cup *subterms'* b)
<proof>

lemma *verts-bgraph*: *verts* (*bgraph* b) = *base-vars* b \cup *subterms'* b
<proof>

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

lemma *finite-subterms'*: *finite* (*subterms'* b)
<proof>

lemma *base-vars-subterms'-disjnt*: *base-vars* b \cap *subterms'* b = $\{\}$
<proof>

lemma *wits-subterms-eq-base-vars-Un-subterms'*:
 assumes *wf-branch* b
 shows *wits-subterms* b = *base-vars* b \cup *subterms'* b
 <proof>

lemma *in-subterms'-if-AT-mem-in-branch*:
 assumes *wf-branch* b
 assumes *AT* (s \in_s t) \in *set* b
 shows s \in *base-vars* b \cup *subterms'* b
 and t \in *base-vars* b \cup *subterms'* b
 <proof>

lemma *in-subterms'-if-AF-mem-in-branch*:
 assumes *wf-branch* b

assumes $AF (s \in_s t) \in set\ b$
shows $s \in base\text{-}vars\ b \cup subterms'\ b$
and $t \in base\text{-}vars\ b \cup subterms'\ b$
 $\langle proof \rangle$

lemma *in-subterms'-if-AT-eq-in-branch:*
assumes $wf\text{-}branch\ b$
assumes $AT (s =_s t) \in set\ b$
shows $s \in base\text{-}vars\ b \cup subterms'\ b$
and $t \in base\text{-}vars\ b \cup subterms'\ b$
 $\langle proof \rangle$

lemma *in-subterms'-if-AF-eq-in-branch:*
assumes $wf\text{-}branch\ b$
assumes $AF (s =_s t) \in set\ b$
shows $s \in base\text{-}vars\ b \cup subterms'\ b$
and $t \in base\text{-}vars\ b \cup subterms'\ b$
 $\langle proof \rangle$

lemma *mem-subterms-fm-last-if-mem-subterms-branch:*
assumes $wf\text{-}branch\ b$
assumes $t \in subterms\ b \neg is\text{-}Var\ t$
shows $t \in subterms\ (last\ b)$
 $\langle proof \rangle$

locale *open-branch =*
fixes $b :: 'a\ branch$
assumes $wf\text{-}branch:$ $wf\text{-}branch\ b$ **and** $bopen:$ $bopen\ b$ **and** $types:$ $\exists v. v \vdash last\ b$
and $infinite\text{-}vars:$ $infinite\ (UNIV :: 'a\ set)$
begin

sublocale *fin-digraph-bgraph:* $fin\text{-}digraph\ bgraph\ b$
 $\langle proof \rangle$

lemma *member-seq-if-cas:*
 $fin\text{-}digraph\text{-}bgraph.cas\ t1\ is\ t2$
 $\implies member\text{-}seq\ t1\ (map\ (\lambda e. tail\ (bgraph\ b)\ e \in_s head\ (bgraph\ b)\ e)\ is)\ t2$
 $\langle proof \rangle$

lemma *member-cycle-if-cycle:*
 $fin\text{-}digraph\text{-}bgraph.cycle\ c$
 $\implies member\text{-}cycle\ (map\ (\lambda e. tail\ (bgraph\ b)\ e \in_s head\ (bgraph\ b)\ e)\ c)$
 $\langle proof \rangle$

sublocale *dag-bgraph:* $dag\ bgraph\ b$
 $\langle proof \rangle$

definition $I :: 'a\ pset\text{-}term \implies hf$ **where**
 $I \equiv SOME\ f. inj\text{-}on\ f\ (subterms\ b)$

$\wedge (\forall p. \text{hcard } (f p) > 2 * \text{card } (\text{base-vars } b \cup \text{subterms}' b))$

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>

lemma

shows *inj-on-I: inj-on I (subterms b)*

and *card-I: hcard (I p) > 2 * card (base-vars b \cup subterms' b)*

<proof>

lemma

shows *inj-on-base-vars-I: inj-on I (base-vars b)*

and *inj-on-subterms'-I: inj-on I (subterms' b)*

<proof>

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

lemma *refl-eq: refl eq*

<proof>

lemma *trans-eq:*

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

<proof>

lemma *sym-eq: sym eq*

<proof>

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

<proof>

lemma *not-dominated-if-pwits:*

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

<proof>

lemma *parents-empty-if-pwits:*

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

<proof>

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

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

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

<proof>

lemma *not-dominated-if-urelems:*

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

shows $\neg s \rightarrow_{\text{bgraph } b} t$

```

⟨proof⟩

lemma parents-empty-if-urelems:
  assumes  $t \in \text{urelems } b$ 
  shows  $\text{parents } (\text{bgraph } b) t = \{\}$ 
  ⟨proof⟩

lemma not-dominated-if-base-vars:
  assumes  $x \in \text{base-vars } b$ 
  shows  $\neg s \rightarrow_{\text{bgraph } b} x$ 
  ⟨proof⟩

lemma parents-empty-if-base-vars:
  assumes  $x \in \text{base-vars } b$ 
  shows  $\text{parents } (\text{bgraph } b) x = \{\}$ 
  ⟨proof⟩

lemma eq-class-sub-subterms:  $\text{eq } \{t\} \subseteq \{t\} \cup \text{subterms } b$ 
  ⟨proof⟩

lemma finite-eq-class:  $\text{finite } (\text{eq } \{x\})$ 
  ⟨proof⟩

lemma finite-I-image-eq-class:  $\text{finite } (I \text{ ` eq } \{x\})$ 
  ⟨proof⟩

context
begin

interpretation realisation  $\text{bgraph } b \text{ base-vars } b \text{ subterms' } b \text{ I eq}$ 
  ⟨proof⟩

lemmas realisation = realisation-axioms

lemma card-realisation:
   $\text{hcard } (\text{realise } t) \leq 2 * \text{card } (\text{subterms } b)$ 
  ⟨proof⟩

lemma I-neq-realise:  $I x \neq \text{realise } y$ 
  ⟨proof⟩

end

sublocale realisation  $\text{bgraph } b \text{ base-vars } b \text{ subterms' } b \text{ I eq}$ 
  rewrites  $(\bigwedge x y. I x \neq \text{realise } y) \equiv \text{Trueprop True}$ 
  and  $\bigwedge P. (\text{True} \implies P) \equiv \text{Trueprop } P$ 
  and  $\bigwedge P Q. (\text{True} \implies \text{PROP } P \implies \text{PROP } Q) \equiv (\text{PROP } P \implies \text{True} \implies$ 
   $\text{PROP } Q)$ 
  ⟨proof⟩

```

lemma *eq-class-singleton-if-pwits*:
assumes $x \in \text{Var}$ ‘ *pwits* b **shows** *eq-class* $x = \{x\}$
 ⟨*proof*⟩

lemma *realise-pwits*:
 $x \in \text{Var}$ ‘ *pwits* $b \implies \text{realise } x = \text{HF } \{I x\}$
 ⟨*proof*⟩

lemma *I-in-realise-if-base-vars[simp]*:
 $s \in \text{base-vars } b \implies I s \in \text{realise } s$
 ⟨*proof*⟩

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*⟩

lemma *AT-mem-branch-wits-subtermsD*:
assumes $AT (s \in_s t) \in \text{set } b$
shows $s \in \text{wits-subterms } b$ $t \in \text{wits-subterms } b$
 ⟨*proof*⟩

lemma *AF-mem-branch-wits-subtermsD*:
assumes $AF (s \in_s t) \in \text{set } b$
shows $s \in \text{wits-subterms } b$ $t \in \text{wits-subterms } b$
 ⟨*proof*⟩

lemma *AT-eq-branch-wits-subtermsD*:
assumes $AT (s =_s t) \in \text{set } b$
shows $s \in \text{wits-subterms } b$ $t \in \text{wits-subterms } b$
 ⟨*proof*⟩

lemma *AF-eq-branch-wits-subtermsD*:
assumes $AF (s =_s t) \in \text{set } b$
shows $s \in \text{wits-subterms } b$ $t \in \text{wits-subterms } b$
 ⟨*proof*⟩

lemma *realisation-if-AT-mem*:
assumes $AT (s \in_s t) \in \text{set } b$
shows $\text{realise } s \in \text{realise } t$
 ⟨*proof*⟩

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

lemma *realisation-if-AT-eq:*

assumes *lin-sat b*

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

shows *realise s = realise t*

<proof>

lemma *realise-base-vars-if-AF-eq:*

assumes *sat b*

assumes $AF (x =_s y) \in set\ b$

assumes $x \in base-vars\ b \vee y \in base-vars\ b$

shows *realise x \neq realise y*

<proof>

lemma *Ex-AT-eq-mem-subterms-last-if-subterms':*

assumes $t \in subterms'\ b$

obtains $t \in subterms (last\ b) - base-vars\ b$

| t' **where** $t' \in subterms (last\ b) - base-vars\ b$

$AT (t =_s t') \in set\ b \vee AT (t' =_s t) \in set\ b$

<proof>

lemma *realisation-if-AF-eq:*

assumes *sat b*

assumes $AF (t1 =_s t2) \in set\ b$

shows *realise t1 \neq realise t2*

<proof>

lemma *realisation-if-AF-mem:*

assumes *sat b*

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

shows *realise s \notin realise t*

<proof>

lemma *realisation-Empty: realise ($\emptyset\ n$) = 0*

<proof>

lemma *realisation-Union:*

assumes *sat b*

assumes $t1 \sqcup_s t2 \in subterms\ b$

shows *realise (t1 \sqcup_s t2) = realise t1 \sqcup realise t2*

<proof>

lemma *realisation-Inter:*

assumes *sat b*

assumes $t1 \sqcap_s t2 \in subterms\ b$

shows *realise (t1 \sqcap_s t2) = realise t1 \sqcap realise t2*

<proof>

lemma *realisation-Diff*:

assumes *sat b*

assumes $s \text{ --}_s t \in \text{subterms } b$

shows $\text{realise } (s \text{ --}_s t) = \text{realise } s \text{ -- } \text{realise } t$

<proof>

lemma *realisation-Single*:

assumes *sat b*

assumes *Single t* $t \in \text{subterms } b$

shows $\text{realise } (\text{Single } t) = \text{HF } \{\text{realise } t\}$

<proof>

lemmas *realisation-simps* =

realisation-Empty realisation-Union realisation-Inter realisation-Diff realisation-Single

end

Coherence

lemma (*in open-branch*) *I_{st}-realisation-eq-realisation*:

assumes *sat b t* $t \in \text{subterms } b$

shows $I_{st} (\lambda x. \text{realise } (\text{Var } x)) t = \text{realise } t$

<proof>

lemma (*in open-branch*) *coherence*:

assumes *sat b* $\varphi \in \text{set } b$

shows $\text{interp } I_{sa} (\lambda x. \text{realise } (\text{Var } x)) \varphi$

<proof>

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*: $(x, x) \notin \text{hmem-rel}^+$

<proof>

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 $(I_{st} M s, I_{st} M t) \in \text{hmem-rel}^+$

<proof>

lemma *bclosed-sound*:

assumes *bclosed b*

shows $\exists \varphi \in \text{set } b. \neg \text{interp } I_{sa} M \varphi$

<proof>

lemma *types-term-lt-if-member-seq*:
includes *no member-ASCII-syntax*
fixes $cs :: 'a \text{ pset-atom list}$
assumes $\forall a \in \text{set } cs. v \vdash a$
assumes *member-seq* $s \ cs \ t \ cs \neq []$
shows $\exists ls \ lt. v \vdash s : ls \wedge v \vdash t : lt \wedge ls < lt$
 $\langle \text{proof} \rangle$

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 $\neg (\text{member-cycle } cs \wedge \text{set } cs \subseteq \text{Atoms } (\text{set } b))$
 $\langle \text{proof} \rangle$

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 $\text{interp } I_{sa} \ M \ \varphi$
 $\langle \text{proof} \rangle$

lemma *bexpands-nowit-sound*:
assumes *bexpands-nowit* $bs' \ b$
assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{sa} \ M \ \psi$
shows $\exists b' \in bs'. \forall \psi \in \text{set } b'. \text{interp } I_{sa} \ M \ \psi$
 $\langle \text{proof} \rangle$

lemma *I_{st}-upd-eq-if-not-mem-vars-term*:
assumes $x \notin \text{vars } t$
shows $I_{st} (M(x := y)) \ t = I_{st} \ M \ t$
 $\langle \text{proof} \rangle$

lemma *I_{sa}-upd-eq-if-not-mem-vars-atom*:
assumes $x \notin \text{vars } a$
shows $I_{sa} (M(x := y)) \ a = I_{sa} \ M \ a$
 $\langle \text{proof} \rangle$

lemma *interp-upd-eq-if-not-mem-vars-fm*:
assumes $x \notin \text{vars } \varphi$
shows $\text{interp } I_{sa} (M(x := y)) \ \varphi = \text{interp } I_{sa} \ M \ \varphi$
 $\langle \text{proof} \rangle$

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 $\exists M. \exists b' \in bs'. \forall \psi \in \text{set } (b' @ b). \text{interp } I_{sa} \ M \ \psi$

<proof>

lemma *bexpands-sound:*

assumes *bexpands* *bs'* *b*

assumes $\bigwedge \psi. \psi \in \text{set } b \implies \text{interp } I_{sa} M \psi$

shows $\exists M. \exists b' \in bs'. \forall \psi \in \text{set } (b' @ b). \text{interp } I_{sa} M \psi$

<proof>

4.2.4 Upper Bounding the Cardinality of a Branch

lemma *Ex-bexpands-wits-if-in-wits:*

assumes *wf-branch* *b*

assumes $x \in \text{wits } b$

obtains *t1 t2 bs b2 b1* **where**

expandss *b* (*b2* @ *b1*) *bexpands-wit* *t1 t2 x bs b1 b2* $\in bs$ *expandss* *b1* [*last b*]

$x \notin \text{wits } b1$ *wits* (*b2* @ *b1*) = *insert* *x* (*wits b1*)

<proof>

lemma *card-wits-ub-if-wf-branch:*

assumes *wf-branch* *b*

shows $\text{card } (\text{wits } b) \leq (\text{card } (\text{subterms } (\text{last } b)))^2$

<proof>

lemma *card-subterms-branch-ub-if-wf-branch:*

assumes *wf-branch* *b*

shows $\text{card } (\text{subterms } b) \leq \text{card } (\text{subterms } (\text{last } b)) + \text{card } (\text{wits } b)$

<proof>

lemma *card-literals-branch-if-wf-branch:*

assumes *wf-branch* *b*

shows $\text{card } \{a \in \text{set } b. \text{is-literal } a\}$

$\leq 2 * (2 * (\text{card } (\text{subterms } (\text{last } b)) + \text{card } (\text{wits } b)))^2$

<proof>

lemma *lexpands-not-literal-mem-subfms-last:*

defines $P \equiv (\lambda b. \forall \psi \in \text{set } b. \neg \text{is-literal } \psi$

$\longrightarrow \psi \in \text{subfms } (\text{last } b) \vee \psi \in \text{Neg } \text{'subfms } (\text{last } b))$

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

assumes *P* *b*

shows *P* (*b'* @ *b*)

<proof>

lemma *bexpands-not-literal-mem-subfms-last:*

defines $P \equiv (\lambda b. \forall \psi \in \text{set } b. \neg \text{is-literal } \psi$

$\longrightarrow \psi \in \text{subfms } (\text{last } b) \vee \psi \in \text{Neg } \text{'subfms } (\text{last } b))$

assumes *bexpands* *bs* *b* *b' ∈ bs* $b \neq []$

assumes *P* *b*

shows *P* (*b'* @ *b*)

<proof>

lemma *expandss-not-literal-mem-subfms-last*:
defines $P \equiv (\lambda b. \forall \psi \in \text{set } b. \neg \text{is-literal } \psi$
 $\quad \rightarrow \psi \in \text{subfms } (\text{last } b) \vee \psi \in \text{Neg } \text{'subfms } (\text{last } b))$
assumes *expandss* $b' b b \neq []$
assumes $P b$
shows $P b'$
 $\langle \text{proof} \rangle$

lemma *card-not-literal-branch-if-wf-branch*:
assumes *wf-branch* b
shows $\text{card } \{\varphi \in \text{set } b. \neg \text{is-literal } \varphi\} \leq 2 * \text{card } (\text{subfms } (\text{last } b))$
 $\langle \text{proof} \rangle$

lemma *card-wf-branch-ub*:
assumes *wf-branch* b
shows $\text{card } (\text{set } b)$
 $\leq 2 * \text{card } (\text{subfms } (\text{last } b)) + 16 * (\text{card } (\text{subterms } (\text{last } b)))^4$
 $\langle \text{proof} \rangle$

4.2.5 The Decision Procedure

locale *mlss-proc* =
fixes *lexpand* :: 'a branch \Rightarrow 'a branch
assumes *lexpands-lexpand*:
 $\neg \text{lin-sat } b \Longrightarrow \text{lexpands } (\text{lexpand } b) b \wedge \text{set } b \subset \text{set } (\text{lexpand } b @ b)$
fixes *bexpand* :: 'a branch \Rightarrow 'a branch set
assumes *bexpands-bexpand*:
 $\neg \text{sat } b \Longrightarrow \text{lin-sat } b \Longrightarrow \text{bexpands } (\text{bexpand } b) b$
begin

function (*domintros*) *mlss-proc-branch* :: 'a branch \Rightarrow bool **where**
 $\neg \text{lin-sat } b$
 $\Longrightarrow \text{mlss-proc-branch } b = \text{mlss-proc-branch } (\text{lexpand } b @ b)$
 $| \llbracket \text{lin-sat } b; \text{bclosed } b \rrbracket \Longrightarrow \text{mlss-proc-branch } b = \text{True}$
 $| \llbracket \neg \text{sat } b; \text{bopen } b; \text{lin-sat } b \rrbracket$
 $\Longrightarrow \text{mlss-proc-branch } b = (\forall b' \in \text{bexpand } b. \text{mlss-proc-branch } (b' @ b))$
 $| \llbracket \text{lin-sat } b; \text{sat } b \rrbracket \Longrightarrow \text{mlss-proc-branch } b = \text{bclosed } b$
 $\langle \text{proof} \rangle$

lemma *mlss-proc-branch-dom-if-wf-branch*:
assumes *wf-branch* b
shows *mlss-proc-branch-dom* b
 $\langle \text{proof} \rangle$

definition *mlss-proc* :: 'a pset-fm \Rightarrow bool **where**
 $\text{mlss-proc } \varphi \equiv \text{mlss-proc-branch } [\varphi]$

lemma *mlss-proc-branch-complete*:

```

fixes  $b :: 'a \text{ branch}$ 
assumes  $\text{wf-branch } b \exists v. v \vdash \text{last } b$ 
assumes  $\neg \text{mlss-proc-branch } b$ 
assumes  $\text{infinite } (\text{UNIV} :: 'a \text{ set})$ 
shows  $\exists M. \text{interp } I_{sa} M (\text{last } b)$ 
<proof>

```

```

lemma  $\text{mlss-proc-branch-sound}$ :
assumes  $\text{wf-branch } b$ 
assumes  $\forall \psi \in \text{set } b. \text{interp } I_{sa} M \psi$ 
shows  $\neg \text{mlss-proc-branch } b$ 
<proof>

```

```

theorem  $\text{mlss-proc-complete}$ :
fixes  $\varphi :: 'a \text{ pset-fm}$ 
assumes  $\neg \text{mlss-proc } \varphi$ 
assumes  $\exists v. v \vdash \varphi$ 
assumes  $\text{infinite } (\text{UNIV} :: 'a \text{ set})$ 
shows  $\exists M. \text{interp } I_{sa} M \varphi$ 
<proof>

```

```

theorem  $\text{mlss-proc-sound}$ :
assumes  $\text{interp } I_{sa} M \varphi$ 
shows  $\neg \text{mlss-proc } \varphi$ 
<proof>

```

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  $\text{subterms-term-list} :: 'a \text{ pset-term} \Rightarrow 'a \text{ pset-term list}$  where
   $\text{subterms-term-list } (\emptyset \ n) = [\emptyset \ n]$ 
|  $\text{subterms-term-list } (\text{Var } i) = [\text{Var } i]$ 
|  $\text{subterms-term-list } (t1 \sqcup_s t2) = [t1 \sqcup_s t2] @ \text{subterms-term-list } t1 @ \text{subterms-term-list } t2$ 
|  $\text{subterms-term-list } (t1 \sqcap_s t2) = [t1 \sqcap_s t2] @ \text{subterms-term-list } t1 @ \text{subterms-term-list } t2$ 
|  $\text{subterms-term-list } (t1 -_s t2) = [t1 -_s t2] @ \text{subterms-term-list } t1 @ \text{subterms-term-list } t2$ 
|  $\text{subterms-term-list } (\text{Single } t) = [\text{Single } t] @ \text{subterms-term-list } t$ 

fun  $\text{subterms-atom-list} :: 'a \text{ pset-atom} \Rightarrow 'a \text{ pset-term list}$  where
   $\text{subterms-atom-list } (t1 \in_s t2) = \text{subterms-term-list } t1 @ \text{subterms-term-list } t2$ 
|  $\text{subterms-atom-list } (t1 =_s t2) = \text{subterms-term-list } t1 @ \text{subterms-term-list } t2$ 

```

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 $\varphi \equiv \text{concat } (\text{map } \text{subterms-atom-list } (\text{atoms-list } \varphi))$

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

lemma *set-subterms-term-list[simp]*:
set (*subterms-term-list* t) = *subterms* t
<proof>

lemma *set-subterms-atom-list[simp]*:
set (*subterms-atom-list* t) = *subterms* t
<proof>

lemma *set-atoms-list[simp]*:
set (*atoms-list* φ) = *atoms* φ
<proof>

lemma *set-subterms-fm-list[simp]*:
set (*subterms-fm-list* φ) = *subterms-fm* φ
<proof>

lemma *set-subterms-branch-list[simp]*:
set (*subterms-branch-list* b) = *subterms* b
<proof>

fun *lexpand-fm1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**
lexpand-fm1 b (And p q) = [[p, q]]
| *lexpand-fm1* b (Neg (Or p q)) = [[Neg p, Neg q]]
| *lexpand-fm1* b (Or p q) =
 (if Neg p \in set b then [[q]] else []) @
 (if Neg q \in set b then [[p]] else [])
| *lexpand-fm1* b (Neg (And p q)) =
 (if p \in set b then [[Neg q]] else []) @
 (if q \in set b then [[Neg p]] else [])
| *lexpand-fm1* b (Neg (Neg p)) = [[p]]
| *lexpand-fm1* b - = []

definition *lexpand-fm* b $\equiv \text{concat } (\text{map } (\text{lexpand-fm1 } b) b)$

lemma *lexpand-fm-if-lexpands-fm*:
lexpands-fm b' b $\implies b' \in \text{set } (\text{lexpand-fm } b)$

<proof>

lemma *lexpands-fm-if-expand-fm1:*

$b' \in \text{set } (\text{expand-fm1 } b \ p) \implies p \in \text{set } b \implies \text{lexpands-fm } b' \ b$

<proof>

lemma *lexpands-fm-if-expand-fm:*

$b' \in \text{set } (\text{expand-fm } b) \implies \text{lexpands-fm } b' \ b$

<proof>

fun *expand-un1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**

expand-un1 b (AF ($s \in_s t$)) =

$[[AF (s \in_s t \sqcup_s t1)]. AF (s' \in_s t1) \leftarrow b, s' = s, t \sqcup_s t1 \in \text{subterms } (\text{last } b)] \ @$

$[[AF (s \in_s t1 \sqcup_s t)]. AF (s' \in_s t1) \leftarrow b, s' = s, t1 \sqcup_s t \in \text{subterms } (\text{last } b)] \ @$

(*case* t of

$t1 \sqcup_s t2 \Rightarrow [[AF (s \in_s t1), AF (s \in_s t2)]]$

| $- \Rightarrow []$)

| *expand-un1* b (AT ($s \in_s t$)) =

$[[AT (s \in_s t \sqcup_s t2)]. t1 \sqcup_s t2 \leftarrow \text{subterms-fm-list } (\text{last } b), t1 = t] \ @$

$[[AT (s \in_s t1 \sqcup_s t)]. t1 \sqcup_s t2 \leftarrow \text{subterms-fm-list } (\text{last } b), t2 = t] \ @$

(*case* t of

$t1 \sqcup_s t2 \Rightarrow (\text{if } AF (s \in_s t1) \in \text{set } b \text{ then } [[AT (s \in_s t2)]] \text{ else } []) \ @$

$(\text{if } AF (s \in_s t2) \in \text{set } b \text{ then } [[AT (s \in_s t1)]] \text{ else } [])$

| $- \Rightarrow []$)

| *expand-un1* $- - = []$

definition *expand-un* $b \equiv \text{concat } (\text{map } (\text{expand-un1 } b) \ b)$

lemma *expand-un-if-lexpands-un:*

$\text{lexpands-un } b' \ b \implies b' \in \text{set } (\text{expand-un } b)$

<proof>

lemma *lexpands-un-if-expand-un1:*

$b' \in \text{set } (\text{expand-un1 } b \ l) \implies l \in \text{set } b \implies \text{lexpands-un } b' \ b$

<proof>

lemma *lexpands-un-if-expand-un:*

$b' \in \text{set } (\text{expand-un } b) \implies \text{lexpands-un } b' \ b$

<proof>

fun *expand-int1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list **where**

expand-int1 b (AT ($s \in_s t$)) =

$[[AT (s \in_s t1 \sqcap_s t)]. AT (s' \in_s t1) \leftarrow b, s' = s, t1 \sqcap_s t \in \text{subterms } (\text{last } b)] \ @$

$[[AT (s \in_s t \sqcap_s t2)]. AT (s' \in_s t2) \leftarrow b, s' = s, t \sqcap_s t2 \in \text{subterms } (\text{last } b)] \ @$

(*case* t of $t1 \sqcap_s t2 \Rightarrow [[AT (s \in_s t1), AT (s \in_s t2)]]$ | $- \Rightarrow []$)

| *expand-int1* b (AF ($s \in_s t$)) =

$[[AF (s \in_s t \sqcap_s t2)]. t1 \sqcap_s t2 \leftarrow \text{subterms-fm-list } (\text{last } b), t1 = t] \ @$

$[[AF (s \in_s t1 \sqcap_s t)]. t1 \sqcap_s t2 \leftarrow \text{subterms-fm-list } (\text{last } b), t2 = t] \ @$

(*case* t of

$t1 \sqcap_s t2 \Rightarrow (if\ AT\ (s \in_s\ t1) \in\ set\ b\ then\ [[AF\ (s \in_s\ t2)]]\ else\ [])\ @$
 $(if\ AT\ (s \in_s\ t2) \in\ set\ b\ then\ [[AF\ (s \in_s\ t1)]]\ else\ [])\ @$
 $| - \Rightarrow [])$
 $| expand-int1\ -\ - = []$

definition $expand-int\ b \equiv concat\ (map\ (expand-int1\ b)\ b)$

lemma $expand-int-if-expands-int$:
 $expands-int\ b'\ b \Longrightarrow b' \in set\ (expand-int\ b)$
 $\langle proof \rangle$

lemma $expands-int-if-expand-int1$:
 $b' \in set\ (expand-int1\ b\ l) \Longrightarrow l \in set\ b \Longrightarrow expands-int\ b'\ b$
 $\langle proof \rangle$

lemma $expands-int-if-expand-int$:
 $b' \in set\ (expand-int\ b) \Longrightarrow expands-int\ b'\ b$
 $\langle proof \rangle$

fun $expand-diff1 :: 'a\ branch \Rightarrow 'a\ pset-fm \Rightarrow 'a\ branch\ list\ \mathbf{where}$
 $expand-diff1\ b\ (AT\ (s \in_s\ t)) =$
 $[[AT\ (s \in_s\ t -_s\ t2)].\ AF\ (s' \in_s\ t2) \leftarrow b,\ s' = s,\ t -_s\ t2 \in subterms\ (last\ b)]\ @$
 $[[AF\ (s \in_s\ t1 -_s\ t)].\ AF\ (s' \in_s\ t1) \leftarrow b,\ s' = s,\ t1 -_s\ t \in subterms\ (last\ b)]\ @$
 $[[AF\ (s \in_s\ t1 -_s\ t)].\ t1 -_s\ t2 \leftarrow subterms-fm-list\ (last\ b),\ t2 = t]\ @$
 $(case\ t\ of\ t1 -_s\ t2 \Rightarrow [[AT\ (s \in_s\ t1),\ AF\ (s \in_s\ t2)]]\ | - \Rightarrow [])$
 $| expand-diff1\ b\ (AF\ (s \in_s\ t)) =$
 $[[AF\ (s \in_s\ t -_s\ t2)].\ t1 -_s\ t2 \leftarrow subterms-fm-list\ (last\ b),\ t1 = t]\ @$
 $(case\ t\ of$
 $t1 -_s\ t2 \Rightarrow (if\ AT\ (s \in_s\ t1) \in\ set\ b\ then\ [[AT\ (s \in_s\ t2)]]\ else\ [])\ @$
 $(if\ AF\ (s \in_s\ t2) \in\ set\ b\ then\ [[AF\ (s \in_s\ t1)]]\ else\ [])$
 $| - \Rightarrow [])$
 $| expand-diff1\ -\ - = []$

definition $expand-diff\ b \equiv concat\ (map\ (expand-diff1\ b)\ b)$

lemma $expand-diff-if-expands-diff$:
 $expands-diff\ b'\ b \Longrightarrow b' \in set\ (expand-diff\ b)$
 $\langle proof \rangle$

lemma $expands-diff-if-expand-diff1$:
 $b' \in set\ (expand-diff1\ b\ l) \Longrightarrow l \in set\ b \Longrightarrow expands-diff\ b'\ b$
 $\langle proof \rangle$

lemma $expands-diff-if-expand-diff$:
 $b' \in set\ (expand-diff\ b) \Longrightarrow expands-diff\ b'\ b$
 $\langle proof \rangle$

fun $expand-single1 :: 'a\ branch \Rightarrow 'a\ pset-fm \Rightarrow 'a\ branch\ list\ \mathbf{where}$
 $expand-single1\ b\ (AT\ (s \in_s\ Single\ t)) = [[AT\ (s =_s\ t)]]$

| *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*:
lexpands-single $b' b \implies b' \in$ *set* (*lexpand-single* b)
 <proof>

lemma *lexpands-single-if-lexpand-single1*:
 $b' \in$ *set* (*lexpand-single1* b l) $\implies l \in$ *set* $b \implies$ *lexpands-single* $b' b$
 <proof>

lemma *lexpands-single-if-lexpand-single*:
 $b' \in$ *set* (*lexpand-single* b) \implies *lexpands-single* $b' b$
 <proof>

fun *lexpand-eq1* :: 'a *branch* \implies 'a *pset-fm* \implies 'a *branch list* **where**
lexpand-eq1 b (AT ($t1 =_s$ $t2$)) =
 $[[AT$ (*subst-tlvl* $t1$ $t2$ a)]. AT $a \leftarrow$ b , $t1 \in$ *tlvl-terms* a] @
 $[[AF$ (*subst-tlvl* $t1$ $t2$ a)]. AF $a \leftarrow$ b , $t1 \in$ *tlvl-terms* a] @
 $[[AT$ (*subst-tlvl* $t2$ $t1$ a)]. AT $a \leftarrow$ b , $t2 \in$ *tlvl-terms* a] @
 $[[AF$ (*subst-tlvl* $t2$ $t1$ a)]. AF $a \leftarrow$ b , $t2 \in$ *tlvl-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 $b' b \implies b' \in$ *set* (*lexpand-eq* b)
 <proof>

lemma *lexpands-eq-if-lexpand-eq1*:
 $b' \in$ *set* (*lexpand-eq1* b l) $\implies l \in$ *set* $b \implies$ *lexpands-eq* $b' b$
 <proof>

lemma *lexpands-eq-if-lexpand-eq*:
 $b' \in$ *set* (*lexpand-eq* b) \implies *lexpands-eq* $b' b$
 <proof>

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*:

$lexpands\ b'\ b \implies b' \in set\ (lexpand\ b)$
 ⟨proof⟩

lemma *lexpands-if-lexpand*:
 $b' \in set\ (lexpand\ b) \implies lexpands\ b'\ b$
 ⟨proof⟩

fun *bexpand-nowit1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow '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' \neg_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 \Rightarrow
 (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 [])
 | - \Rightarrow [])
 | *bexpand-nowit1* b - = []

definition *bexpand-nowit* b \equiv concat (map (*bexpand-nowit1* b) b)

lemma *bexpand-nowit-if-bexpands-nowit*:
 $bexpands\ nowit\ bs'\ b \implies bs' \in set\ 'set\ (bexpand\ nowit\ b)$
 ⟨proof⟩

lemma *bexpands-nowit-if-bexpand-nowit1*:
 $bs' \in set\ 'set\ (bexpand\ nowit1\ b\ l) \implies l \in set\ b \implies bexpands\ nowit\ bs'\ b$
 ⟨proof⟩

lemma *bexpands-nowit-if-bexpand-nowit*:
 $bs' \in set\ 'set\ (bexpand\ nowit\ b) \implies bexpands\ nowit\ bs'\ b$
 ⟨proof⟩

definition *name-subterm* $\varphi \equiv index\ (subterms\ fm\ list\ \varphi)$

lemma *inj-on-name-subterm-subterms*:
 $inj\ on\ (name\ subterm\ \varphi)\ (subterms\ \varphi)$
 ⟨proof⟩

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 $t \in$ subterms φ
shows *urelem* $\varphi t \longleftrightarrow$ *urelem-code* φt
 <proof>

fun *bexpand-wit1* :: ('a::fresh0) branch \Rightarrow 'a pset-fm \Rightarrow 'a branch list list **where**
bexpand-wit1 b (AF ($t1 =_s t2$)) =
 (if $t1 \in$ subterms (last b) $\wedge t2 \in$ subterms (last b) \wedge
 ($\forall t \in$ set b . case t of AT ($x \in_s t1'$) $\Rightarrow t1' = t1 \longrightarrow$ AF ($x \in_s t2$) \notin set b |
 - \Rightarrow True) \wedge
 ($\forall t \in$ set b . case t of AT ($x \in_s t2'$) $\Rightarrow t2' = t2 \longrightarrow$ AF ($x \in_s t1$) \notin set b |
 - \Rightarrow True) \wedge
 \neg *urelem-code* (last b) $t1 \wedge \neg$ *urelem-code* (last b) $t2$
 then
 (let $x =$ fresh0 (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* $b \equiv$ concat (map (*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
 AT ($x \in_s t'$) $\Rightarrow t' = t1 \longrightarrow$ AF ($x \in_s t2$) \notin set b
 | - \Rightarrow True)
 <proof>

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>

lemma *bexpand-wit-if-bexpands-wit*:
assumes *bexpands-wit* $t1 t2$ (fresh0 (vars b)) $bs' b$
shows $bs' \in$ set ' set (*bexpand-wit* b)
 <proof>

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>

lemma *beexpands-wit-if-beexpand-wit*:

$bs' \in \text{set } ' \text{ set } (\text{beexpand-wit } b) \implies (\exists t1 \ t2 \ x. \text{beexpands-wit } t1 \ t2 \ x \ bs' \ b)$
(proof)

definition $\text{beexpand } b \equiv \text{beexpand-nowit } b \ @ \ \text{beexpand-wit } b$

lemma *beexpands-if-beexpand*:

$bs' \in \text{set } ' \ \text{set } (\text{beexpand } b) \implies \text{beexpands } bs' \ b$
(proof)

lemma *Not-beexpands-if-beexpand-empty*:

assumes $\text{beexpand } b = []$
shows $\neg \text{beexpands } bs' \ b$
(proof)

lemma *lin-sat-code*:

$\text{lin-sat } b \longleftrightarrow \text{filter } (\lambda b'. \neg \text{set } b' \subseteq \text{set } b) (\text{lexpand } b) = []$
(proof)

lemma *sat-code*:

$\text{sat } b \longleftrightarrow \text{lin-sat } b \wedge \text{beexpand } b = []$
(proof)

fun *bclosed-code1* :: 'a branch \Rightarrow 'a pset-fm \Rightarrow bool **where**

$\text{bclosed-code1 } b \ (\text{Neg } \varphi) \longleftrightarrow$
 $\varphi \in \text{set } b \vee$
 $(\text{case } \varphi \text{ of } \text{Atom } (t1 =_s t2) \Rightarrow t1 = t2 \mid - \Rightarrow \text{False})$
 $\mid \text{bclosed-code1 } b \ (\text{AT } (- \in_s \emptyset -)) \longleftrightarrow \text{True}$
 $\mid \text{bclosed-code1 } - \ - \longleftrightarrow \text{False}$

definition $\text{bclosed-code } b \equiv (\exists t \in \text{set } b. \text{bclosed-code1 } b \ t)$

lemma *bclosed-code-if-bclosed*:

assumes $\text{bclosed } b \ \text{wf-branch } b \ v \vdash \text{last } b$
shows $\text{bclosed-code } b$
(proof)

lemma *bclosed-if-bclosed-code1*:

$\text{bclosed-code1 } b \ l \implies l \in \text{set } b \implies \text{bclosed } b$
(proof)

lemma *bclosed-if-bclosed-code*:

$\text{bclosed-code } b \implies \text{bclosed } b$
(proof)

lemma *bclosed-code*:

assumes $\text{wf-branch } b \ v \vdash \text{last } b$
shows $\text{bclosed } b \longleftrightarrow \text{bclosed-code } b$
(proof)

definition *lexpand-safe* $b \equiv$
case filter $(\lambda b'. \neg \text{set } b' \subseteq \text{set } b) (\text{lexpand } b)$ of
 $b' \# bs' \Rightarrow b'$
 $| \square \Rightarrow \square$

lemma *lexpands-lexpand-safe*:
 $\neg \text{lin-sat } b \Longrightarrow \text{lexpands } (\text{lexpand-safe } b) b \wedge \text{set } b \subset \text{set } (\text{lexpand-safe } b @ b)$
 $\langle \text{proof} \rangle$

lemma *wf-branch-lexpand-safe*:
assumes *wf-branch* b
shows *wf-branch* $(\text{lexpand-safe } b @ b)$
 $\langle \text{proof} \rangle$

definition *bexpand-safe* $b \equiv$
case bexpand b of
 $bs' \# bss' \Rightarrow bs'$
 $| \square \Rightarrow [\square]$

lemma *bexpands-bexpand-safe*:
 $\neg \text{sat } b \Longrightarrow \text{lin-sat } b \Longrightarrow \text{bexpands } (\text{set } (\text{bexpand-safe } b)) b$
 $\langle \text{proof} \rangle$

lemma *wf-branch-bexpand-safe*:
assumes *wf-branch* b
shows $\forall b' \in \text{set } (\text{bexpand-safe } b). \text{wf-branch } (b' @ b)$
 $\langle \text{proof} \rangle$

interpretation *mlss-naive*: *mlss-proc* *lexpand-safe* *set o bexpand-safe*
 $\langle \text{proof} \rangle$

lemma *types-pset-fm-code*:
 $(\exists v. v \vdash \varphi) \longleftrightarrow \text{solve-constraints } \varphi \neq \text{None}$
 $\langle \text{proof} \rangle$

fun *foldl-option where*
foldl-option $f a \square = \text{Some } a$
 $| \text{foldl-option } f - (\text{None} \# -) = \text{None}$
 $| \text{foldl-option } f a (\text{Some } x \# xs) = \text{foldl-option } f (f a x) xs$

lemma *monotone-fold-option-conj*[*partial-function-mono*]:
monotone $(\text{list-all2 } \text{option-ord}) \text{option-ord } (\text{foldl-option } f a)$
 $\langle \text{proof} \rangle$

lemma *monotone-map*[*partial-function-mono*]:
assumes *monotone* $(\text{list-all2 } \text{option-ord}) \text{ordb } B$
shows *monotone option.le-fun* $\text{ordb } (\lambda h. B (\text{map } h xs))$
 $\langle \text{proof} \rangle$

```

partial-function (option) mlss-proc-branch-partial
  :: ('a::fresh0) branch  $\Rightarrow$  bool option where
    mlss-proc-branch-partial b =
      (if  $\neg$  lin-sat b then mlss-proc-branch-partial (lexpand-safe b @ b)
       else if bclosed-code b then Some True
       else if  $\neg$  sat b then
         foldl-option ( $\wedge$ ) True (map mlss-proc-branch-partial (map ( $\lambda b'$ . b' @ b)
           (bexpand-safe b)))
         else Some (bclosed-code b))

```

```

lemma mlss-proc-branch-partial-eq:
  assumes wf-branch b v  $\vdash$  last b
  shows mlss-proc-branch-partial b = Some (mlss-naive.mlss-proc-branch b)
    (is ?mlss-part b = Some (?mlss b))
  <proof>

```

```

definition mlss-proc-partial ( $\varphi$  :: nat pset-fm)  $\equiv$ 
  if solve-constraints  $\varphi$  = None then None else mlss-proc-branch-partial [ $\varphi$ ]

```

```

lemma mlss-proc-partial-eq-None:
  mlss-proc-partial  $\varphi$  = None  $\implies$  ( $\nexists$  v. v  $\vdash$   $\varphi$ )
  <proof>

```

```

lemma mlss-proc-partial-complete:
  assumes mlss-proc-partial  $\varphi$  = Some False
  shows  $\exists$  M. interp  $I_{sa}$  M  $\varphi$ 
  <proof>

```

```

lemma mlss-proc-partial-sound:
  assumes mlss-proc-partial  $\varphi$  = Some True
  shows  $\neg$  interp  $I_{sa}$  M  $\varphi$ 
  <proof>

```

```

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.