Towards a Verified Tableau Prover for a Quantifier-Free Fragment of Set Theory

Lukas Stevens[0000−0003−0222−6858]

Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany
`lukas.stevens@in.tum.de`

Abstract Using Isabelle/HOL, we verify the state-of-the-art decision procedure for multi-level syllogistic with singleton (**MLSS** for short), which is a quantifier-free fragment of set theory. We formalise its syntax and semantics as well as 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.

Keywords: Decision procedures · Tableau · Proof assistants · Set theory

1 Introduction

In Isabelle/HOL, there are specialised procedures for dealing with e.g. natural numbers, linear arithmetic, and metric spaces. Some of these procedures have been verified in Isabelle/HOL such as a procedure for Presburger arithmetic [12] that was later extended to mixed real-integer arithmetic [11]. This procedure, though, uses reflection to work on goals in Isabelle/HOL, which, during execution, either sacrifices speed by going through the simplifier or requires trusting the code generator. More recently, Stevens and Nipkow [26] presented a verified decision procedure for orders that produces certificates. This approach offers efficient execution by using generated code as well as soundness because the certificates are replayed through Isabelle's inference kernel.

The focus of this paper is another ubiquitous structure in mathematics, namely sets. To the best of our knowledge, we present the first formally verified decision procedure for (a fragment of) set theory. In particular, we consider a quantifier-free fragment which Cantone and Zarba [9] call multi-level syllogistic with singleton (**MLSS**). The fragment includes the usual set operations of union, intersection, difference, membership, equality and, in addition, it allows the construction of singleton sets.

Since **MLSS** admits a tableau calculus, the generation of certificates will be straightforward. Like with the aforementioned order solver, this paves the way towards an integration of the decision procedure into Isabelle, adding to its growing body of verified decision procedures.

1.1 Contributions

We present a formalisation in Isabelle/HOL of a tableau calculus for **MLSS** due to Cantone and Zarba [9][7, Chapter 14]. We prove soundness and completeness of the calculus and give an abstract specification of a decision procedure that applies the rules of the calculus exhaustively. To obtain total correctness of the procedure, we prove its termination. In order to obtain an executable procedure, we also give a naive refinement to a more concrete specification from which code can be generated. The formalisation initially follows the paper but gives a more thorough account of some important details:

- We deliver the omitted proof of Lemma 2 in the paper [9], which is a key building block for the completeness proof of the calculus.
- The formal proof of completeness lead to the discovery that the calculus was missing a rule for eliminating double negation.
- We derive an explicit upper bound for the number of formulae in a branch of the tableau.

In the context of Isabelle/HOL, there is one important aspect that requires us to modify the calculus in the paper: the calculus works under the assumption that every variable is a set; however, this is not the case in Isabelle/HOL, e.g. consider the expression `n ∈ A` where `n` is a natural number. In order to deal with these urelements, we extend the calculus with a light-weight type system and a verified inference algorithm that identifies the urelements.

The modification of the calculus required non-trivial changes to the completeness proof. Here, the formalisation was instrumental because Isabelle immediately revealed which proofs had been broken. This illustrates the usefulness of ITPs for developing logic calculi: they allow us to confidently make modifications without compromising correctness.

All in all, the formalisation amounts to over 6000 lines of theory. It is available online [25].

1.2 Related Work

Since the literature on decidable fragments of set theory is vast, we only focus on **MLSS** here. The fragment was first shown to be decidable by Ferro et al. [14]. Subsequent work [6] found the decision problem to be **NP**-complete. To obtain a practical decision procedure, Cantone [4] proposed a tableau calculus, which was later improved by Beckert and Hartmer [1]. Both of these procedures construct a model during execution that is used to guide the proof search. Beckert and Hartmer also cover an extension of the calculus with uninterpreted functions, which was revisited by Cantone and Zarba [10] who avoided the construction of a model during execution. In this paper, we consider a version of the latter procedure due to Cantone and Zarba [9] that is specialised to **MLSS** and where the branching rules of the calculus are set up in a way to guarantee mutual exclusivity of the branches. Later extensions of the calculus added certain interpreted functions such as monotone functions [8] and the inverse of a function [5].

The second extension notably includes the Cartesian product. Those extensions, though, did not improve upon the tableau calculus for **MLSS**.

There is a large body of work at the intersection of ITPs and tableau methods but to keep with the theme of the paper we only consider formalisations of correctness here. For first-order logic, there are abstract completeness proofs using the *Beth-Hintikka style* of possibly infinite derivation trees [3] as well as the *Henkin style* of maximally consistent sets [17]. Both are abstract enough to be instantiated with a wide range of concrete calculi. A more concrete formalisation [19] verifies a sequent calculus for first-order logic whose completeness proof is via a translation to semantic tableau.

Beyond completeness, we target decidability which is more attainable for propositional logic. There is a verified tableau calculus for the modal logic S5 [2] in Lean and one for hybrid logic [18] in Isabelle/HOL. Both of these formalisation do not prove termination but there is a formalisation of a tableau calculus for the temporal logic CTL in Coq [13] that does.

1.3 Notation

Isabelle/HOL [21] conforms to everyday mathematical notation for the most part. We establish notation and in particular some essential data types together with their primitive operations that are specific to Isabelle/HOL.

We write `t :: 'a` to specify that the term `t` has the type `'a` and `'a ⇒ 'b` for the space of total functions from type `'a` to type `'b`.

Sets with elements of type `'a` have the type `'a set`. The cardinality of a set `A` is denoted by `|A|` and the image of `A` under `f` by `f ‘ A`.

We use `'a list` to describe the type of lists, which are constructed using the empty list `[]` constructor or the infix cons constructor `#`, and are appended with the infix operator `@`. The function `set` converts a list into a set.

We remark that $\longleftrightarrow$ is equivalent to $=$ on the type of Booleans `bool` and $\equiv$ is definitional equality of the meta-logic of Isabelle/HOL, which is called Isabelle/Pure. Meta-implication is denoted by $\Longrightarrow$ and a chain of implications $A_1 \Longrightarrow \cdots \Longrightarrow A_k \Longrightarrow C$ can be abbreviated by $[\![A_1; \ldots; A_k]\!] \Longrightarrow C$.

2 Syntax and Semantics of MLSS

2.1 Syntax

At the heart of **MLSS**, we have the type of set terms which is the disjoint union of the empty set and variables as well as the operations union, intersection, set difference, and the singleton set represented by the constructor `Single`. We keep the type of variables abstract by making it a parameter of the set term data type. The only restriction on the type of variables is that it needs to be infinite. Isabelle/HOL's data type package automatically defines a function that gives us the set of variables in a set term, which we name `vars`. In what follows, we will overload the function `vars` to also work on set atoms, formulae, and branches.

```
datatype (vars: 'a) pset_term =
  ∅ | Var 'a | Single ('a pset_term)
| 'a pset_term ⊔s 'a pset_term | 'a pset_term ⊓s 'a pset_term
| 'a pset_term -s 'a pset_term
```

We can combine two set terms to form a set atom by using the membership or the equality operator.

```
datatype (vars: 'a) pset_atom =
  'a pset_term ∈s 'a pset_term | 'a pset_term =s 'a pset_term
```

With the above operators we can also represent the subset operator $\sqsubseteq_s$ and enumerate finite sets: `s` $\sqsubseteq_s$ `t` is equivalent to `s` $\sqcup_s$ `t` $=_s$ `t` and a finite set of elements $\{t_1,\ldots,t_k\}$ can be expressed by `Single` $t_1 \sqcup_s \ldots \sqcup_s$ `Single` t_k.

We use the propositional fragment of formulae due to Nipkow [20] with set atoms as propositional atoms to form the quantifier-free fragment **MLSS** of set theory.

```
datatype (atoms: 'a) fm =
  A 'a | ¬ ('a fm) | 'a fm ∧ 'a fm | 'a fm ∨ 'a fm
```

```
type_synonym 'a pset_fm = 'a pset_atom fm
```

We will often drop the atom constructor `A` to reduce clutter. Additionally, we use `s` $\notin_s$ `t` and `s` $\neq_s$ `t` to denote `¬ A (s` $\in_s$ `t)` and `¬ A (s` $=_s$ `t)`, respectively.

Similarly to `vars`, we get the function `atoms :: 'a fm ⇒ 'a set` for free which retrieves all set atoms in a formula. We combine these functions to extract all the variables occurring in a set formula.

```
definition vars φ ≡ ⋃(vars ` atoms φ)
```

Likewise, we fix the constant `subterms :: 'b ⇒ 'a pset_term set`. We overload this constant to return the set terms that are subterms of a set term, set atom, or formula, respectively. Lastly, we compute the subformulae of a formula with the function `subfms :: 'a fm ⇒ 'a fm set`. The functions `subterms` and `subfms` are defined in the way (cf. Appendix A).

2.2 Semantics

The original paper [9] bases the semantics of **MLSS** on the von Neumann hierarchy of sets $\mathcal{V}$. While there is an entry [24] in the *Archive of Formal Proofs* (AFP) that axiomatises this hierarchy, we instead use the hierarchy of *hereditarily finite sets* (HF sets) which fulfil all the same axioms as $\mathcal{V}$ — that is, the axioms of ZF — except for the axiom of infinity. In particular, the membership relation is well-founded. The HF sets are sufficient for our purposes as **MLSS** is not powerful enough to force infinite models [7, Chapter 14.2]. In contrast to $\mathcal{V}$, the HF sets are directly representable in Isabelle/HOL, and indeed, there is an AFP entry [23] that formalises them. The entry defines a type `hf` that comes with the following functionality:

– The function `HF :: hf set ⇒ set` that converts a finite set of HF sets into an HF set.
– The usual set operations such as equality (=), membership (∈), union (⊔), intersection (⊓), and difference (−) are defined.
– Finally, the empty set coincides with the ordinal 0, so it is denoted by `0 :: hf`.

Equipped with the above, we define the interpretation functions

– $\mathtt{I}_{\mathtt{st}}$ `:: ('a ⇒ hf) ⇒ 'a pset_term ⇒ hf` and
– $\mathtt{I}_{\mathtt{sa}}$ `:: ('a ⇒ hf) ⇒ 'a pset_atom ⇒ hf`

in the standard way, i.e. by mapping each syntactic construct to the corresponding operation on HF sets and interpreting variables with respect to a given valuation function `M :: 'a ⇒ hf`. For the concrete definition we refer to formalisation (cf. Appendix B).

We write $\mathtt{M} \models \phi$ for the judgement that the formula ϕ holds under the valuation function `M`. The implementation of $\models$ coincides with the interpretation function of Nipkow [20]. As usual, a formula ϕ is called *satisfiable* if there exists a model `M` with $\mathtt{M} \models \phi$. Otherwise, ϕ is called *unsatisfiable*.

3 A Tableau Calculus for MLSS

We formalise the tableau calculus for **MLSS** as described by Cantone and Zarba [9]. Inspired by the formalisation of a tableau calculus for hybrid logic by From [16], we simply use a list to represent a branch of the tableau tree. Note that formulae are added to the front of the list during branch expansion, so `last b` for a branch `b` is always the formula that we are trying to disprove with the tableau. We sometimes call this formula the *initial formula*.

```
type_synonym 'a branch = 'a pset_fm list
```

The functions `vars` and `subterms` are lifted to branches in the expected way.

In the standard tableau calculus for propositional logic as Fitting [15] describes it, a branch is called *closed* if it contains both the negation of a formula and the formula itself; conversely, it is called *open* if it is not closed. For **MLSS**, we extend the notion of closedness with three additional rules; the first two are straightforward while the last one states that a branch is closed when the branch contains a membership cycle $\mathtt{t}_0 \in_{\mathtt{s}} \mathtt{t}_1,\ \mathtt{t}_1 \in_{\mathtt{s}} \mathtt{t}_2,\ \ldots,\ \mathtt{t}_{\mathtt{k}} \in_{\mathtt{s}} \mathtt{t}_0$.

```
inductive bclosed :: 'a branch ⇒ bool where
  ⟦ φ ∈ set b; ¬ φ ∈ set b ⟧ ⟹ bclosed b
| (t ∈s ∅) ∈ set b ⟹ bclosed b
| (t ≠s t) ∈ set b ⟹ bclosed b
| ⟦ member_cycle cs; set cs ⊆ set b⟧ ⟹ bclosed b
```

```
abbreviation bopen b ≡ ¬ bclosed b
```

A tableau is called *closed* if all of its branches are closed.

Table 1. Linear expansion rules. We omit the rules for $\sqcap_s$ and $-_s$ as they coincide with the original paper (cf. Appendix D).

Propositional Rules		Rules for $\sqcup_s$	
$p \wedge q$	$\Longrightarrow p,\ q$	$s \notin_s t_1 \sqcup_s t_2$	$\Longrightarrow s \notin_s t_1,\ s \notin_s t_2$
$\neg(p \vee q)$	$\Longrightarrow \neg p,\ \neg q$	$s \in_s t_1$	$\Longrightarrow s \in_s t_1 \sqcup_s t_2$
$p \vee q,\ \neg p$	$\Longrightarrow q$	$s \in t_2$	$\Longrightarrow s \in_s t_1 \sqcup_s t_2$
$p \vee q,\ \neg q$	$\Longrightarrow p$	$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_1$	$\Longrightarrow s \in_s t_2$
$\neg(p \wedge q),\ p$	$\Longrightarrow \neg q$	$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_2$	$\Longrightarrow s \in_s t_1$
$\neg(p \wedge q),\ q$	$\Longrightarrow \neg p$	$s \notin_s t_1,\ s \notin_s t_2$	$\Longrightarrow s \notin_s t_1 \sqcup_s t_2$
$\neg(\neg p)$	$\Longrightarrow p$		
Rules for $\sqcap_s$		**Rules for $-_s$**	
$\vdots$		$\vdots$	
Rules for `Single`		**Rules for $=_s$**	
	$\Longrightarrow s \in_s \texttt{Single}\ s$	$t_1 =_s t_2,\ l$	$\Longrightarrow l\{t_2/t_1\}$
$s \in_s \texttt{Single}\ t$	$\Longrightarrow s =_s t$	$t_1 =_s t_2,\ l$	$\Longrightarrow l\{t_1/t_2\}$
$s \notin_s \texttt{Single}\ t$	$\Longrightarrow s \neq_s t$	$s_1 \in_s t,\ s_2 \notin_s t$	$\Longrightarrow s_1 \neq_s s_2$

3.1 Linear Expansion Rules

The calculus considers two kinds of branch expansion rules: *linear* and *branching* rules. As the name suggests, branching rules lead to the creation of new branches in the tableau while linear rules only extend a branch `b` with new formulae `b' = [`$\psi_1,\ldots,\psi_n$`]`, which we denote by `b'` $\rhd$ `b`. Table 1 shows the linear expansion rules. Note that in the first two rules for $=_s$, `l` is a literal occurring in the branch. Furthermore, the term-for-term substitution `l{s/t}` is restricted to replace `t` only if it occurs as a top-level set terms of `l`. As an example, `u` and $s \sqcap_s t$ are top-level terms of $u \in_s s \sqcap_s t$ but `s` and `t` are not.

The more crucial restriction of the linear rules is that that no new subterm may be created by their application; for instance, the second rule for $\sqcup_s$ is $s \in_s t_1 \Longrightarrow s \in_s t_1 \sqcup_s t_2$, which formally represents

$$(s \in_s t_1) \in \texttt{set b} \Longrightarrow [s \in_s t_1 \sqcup_s t_2] \rhd b,$$

may only be used under the condition $t_1 \sqcup_s t_2 \in$ `subterms (last b)`. The purpose of this restriction is to prevent unbounded expansion of the branch. In fact, we give an explicit upper bound for the number of formulae in a branch in section 7.

Due to boundedness, repeated expansion with linear rules eventually results in a *linearly saturated* branch, i.e. a branch where no application of linear rules would produce new formulae.

```
definition lin_sat b ≡ ∀b'. b' ▷ b ⟶ set b' ⊆ set b
```

Table 2. Branching expansion rules. We write ϕ for `last b` here. The omitted rules coincide with the original paper (cf. Appendix E).

Rule	Precondition	Subsumption condition
$\dfrac{}{\mathtt{p} \mid \neg\,\mathtt{p}}$	$\mathtt{p} \lor \mathtt{q} \in \mathtt{set\ b}$	$\mathtt{p} \in \mathtt{set\ b} \lor$ $\neg\,\mathtt{p} \in \mathtt{set\ b}$
$\dfrac{}{\mathtt{s} \in_s \mathtt{t}_1 \mid \mathtt{s} \notin_s \mathtt{t}_1}$	$(\mathtt{s} \in_s \mathtt{t}_1 \sqcup_s \mathtt{t}_2) \in \mathtt{set\ b}$ $\mathtt{t}_1 \sqcup_s \mathtt{t}_2 \in \mathtt{subterms}\ \phi$	$(\mathtt{s} \in_s \mathtt{t}_1) \in \mathtt{set\ b}$ $\lor\ (\mathtt{s} \notin_s \mathtt{t}_1) \in \mathtt{set\ b}$
$\vdots$	$\vdots$	$\vdots$
$\dfrac{}{\begin{matrix}\mathtt{Var\ x} \in_s \mathtt{t}_1 \\ \mathtt{Var\ x} \notin_s \mathtt{t}_2\end{matrix} \,\Bigg\vert\, \begin{matrix}\mathtt{Var\ x} \notin_s \mathtt{t}_1 \\ \mathtt{Var\ x} \in_s \mathtt{t}_2\end{matrix}}$	$(\mathtt{t}_1 \neq_s \mathtt{t}_2) \in \mathtt{set\ b}$ $\mathtt{t}_1 \in \mathtt{subterms}\ \phi$ $\mathtt{t}_2 \in \mathtt{subterms}\ \phi$ $\mathtt{x} \notin \mathtt{vars\ b}$	$\exists \mathtt{s}.\ (\mathtt{s} \in_s \mathtt{t}_1) \in \mathtt{set\ b}$ $\land\ (\mathtt{s} \notin_s \mathtt{t}_2) \in \mathtt{set\ b}$ $\lor$ $\exists \mathtt{s}.\ (\mathtt{s} \notin_s \mathtt{t}_1) \in \mathtt{set\ b}$ $\land\ (\mathtt{s} \in_s \mathtt{t}_2) \in \mathtt{set\ b}$

Finally, we remark that the original paper [9] is missing the last propositional rule dealing with double negation. This rule is required for completeness, though, considering that the branch `[¬¬¬p, p, ¬¬¬p ∧ p]` is saturated — neither linear nor branching rules apply — and open, but there clearly is no model for the initial formula `¬ ¬ ¬ p ∧ p`.

3.2 Branching Rules

After running out of linear rules to apply, only the branching rules shown in Table 2 remain. A rule is applicable if its *precondition* is met and, to prevent unnecessary branching, it is not subsumed as indicated by the *subsumption condition.* These rules create multiple branches in the tableau, so we represent the different possibilities `bs'` to expand a branch `b` as a set and write `bs' ⊳ b`. Accordingly, we get a new branch `b' @ b` in the tableau for each `b' ∈ set bs'`.

A linearly saturated branch where no further branching is possible is called a *saturated* branch.

```
definition sat b ≡ lin_sat b ∧ (∄bs'. bs' ⊳ b)
```

Note that even branching rules are defined such that they never create new subterms with exception of the last rule, which adds a new variable to the branch. These variables serve to manifest an inequality; hence, we call them *witnesses.*

```
definition wits b ≡ vars b - vars (last b)
```

4 A Decision Procedure for MLSS

The mechanics of the decision procedure are typical for a procedure based on a tableau calculus: it decides the satisfiability of a given formula ϕ by determining whether the formula has a closed tableau. More specifically, it initialises the tableau with the singleton branch [ϕ] and checks whether this branch can be expanded to a closed tableau.

We only discuss the abstract specification here and refer the reader to the formalisation for the executable specification. The implementation uses a couple of features of Isabelle/HOL's function package: instead of defining the function via pattern matching, we specify the equations of the function as conditional rewrite rules. This requires us to prove that the assumptions of the equations are non-overlapping, which is done by automation. The other concern is that Isabelle/HOL requires functions to be total, so a recursive function needs to terminate in order for it to be well-defined; nevertheless, the termination proof is separated from the definition of the function for modularity. The function package maintains the soundness of the definition by introducing a so-called domain predicate `mlss_proc_branch_dom` which characterises the arguments for which the function terminates. Each equation of the function is guarded by an assumption that the predicate holds for the argument. In section 7, we will show that the domain predicate holds for the context that `mlss_proc_branch` is called in. Before we go into more detail on how the termination is proved, we discuss its definition as shown below.

```
function mlss_proc_branch :: 'a branch ⇒ bool where
  ¬ lin_sat b ⟹ mlss_proc_branch b =
  mlss_proc_branch ((SOME b'. b' ▷ b ∧
                              set b ⊂ set (b' @ b)) @ b)
| ⟦ lin_sat b; bclosed b ⟧ ⟹ mlss_proc_branch b = True
| ⟦ ¬ sat b; bopen b; lin_sat b ⟧ ⟹ mlss_proc_branch b =
  (∀b' ∈ (SOME bs. bs ⊳ b). mlss_proc_branch (b' @ b))
| ⟦ lin_sat b; sat b ⟧ ⟹ mlss_proc_branch b = bclosed b

definition mlss_proc :: 'a pset_fm ⇒ bool where
  mlss_proc φ ≡ mlss_proc_branch [φ]
```

The purpose of the function is to determine whether a given branch can be expanded to a closed tableau. As stated before, we first use linear expansion rules in order to prevent premature branching; to this end, we recursively expand the branch with linear rules until the branch is linearly saturated. Note that we use Hilbert's ε-operator in the form of `SOME` to choose some rule that actually adds new formulae to the branch. As soon as the branch is linearly saturated, we terminate if the branch is closed as shown in the second equation. Otherwise, we choose an applicable branching rule and recursively check whether all newly created branches can be closed. The final equation applies once no further branch expansion is possible, in which case we just test for closedness of the branch.

The procedure `mlss_proc` then calls `mlss_proc_branch` with a singleton branch [ϕ] to determine the satisfiability of a given formula ϕ.

This means that `mlss_proc_branch` is only applied to branches that result from applying the expansion rules. We call this kind of branches *well-formed*.

In the definition below, the expression `b' ▷* b` denotes that `b'` is one of the branches that results from applying (potentially zero) expansion rules to `b`.

```
definition wf_branch b ≡ ∃ϕ. b ▷* [ϕ]
```

In particular, we will use this notion in section 7 to state an upper bound for the cardinality of well-formed branches, with which we justify the termination of the decision procedure. Before we come to that, though, we prove soundness and completeness in Section 6 and 5, respectively. In section 7, we also show that both properties easily transfer to `mlss_proc` which, together with termination, establishes that it is a decision procedure.

5 Completeness of the Calculus

For completeness of the calculus, we need to show that every unsatisfiable formula has a closed tableau or, conversely, that the formula is satisfiable if there is a saturated and open branch in the tableau. To facilitate inductive reasoning, we show a stronger statement by constructing a model `M` such that `M` $\models$ ϕ for all ϕ $\in$ `set b`. At the core of the model, there is a *realisation* function that maps set terms to sets of type `hf`. A subset of the witnesses, which we call *pure* witnesses, receives special treatment from the realisation function for reasons that will become apparent in subsection 5.1. The collection of set terms of a branch can thus be partitioned into two collections as defined below.

```
definition pwits :: 'a branch ⇒ 'a set where
  pwits b ≡ {c ∈ wits b. ∀t ∈ subterms (last b).
               AT (Var c =s t) ∉ set b ∧ AT (t =s Var c) ∉ set b}

definition subterms' :: 'a branch ⇒ 'a pset_term set where
  subterms' b ≡ subterms (last b) ∪ Var ‘ (wits b - pwits b)
```

We aim to construct a syntactic model that is derived from the membership literals `s ∈s t` in the branch. To this end, we construct a graph whose vertices are the disjoint union of the sets above and there is an edge from `s` to `t` in the graph if, and only if, `s ∈s t` is in `b`. Note that we use Noschinski's graph library [22] that represents a graph as a record of vertices, arcs (directed edges), and two functions `tail` and `head` that map an arc to its source and target vertex, respectively.

```
definition bgraph b ≡ let vs = Var ‘ pwits b ∪ subterms' b
  in ⦇ verts = vs, arcs = {(s, t). (s ∈s t) ∈ set b},
       tail = fst, head = snd ⦈
```

The realisation function is defined relative to this graph. As mentioned before, the pure witnesses are treated differently than the rest of the set terms by the realisation function. Terms in the latter set are evaluated in accordance to the structure of the graph, i.e. the realisation of a vertex is defined as the union of the realisations of the parent vertices. For the former set, we choose a function `I` that assigns the pure witnesses pairwise distinct sets with cardinality greater than that of the vertices. We can always choose such a function since we assume an infinite universe of variables. Then, we return the singleton set `HF {I x}`, which, together with the cardinality constraint, guarantees that realisations are distinct between pure witnesses themselves as well as between pure witnesses and set terms. The notation `u →G s` in the definition below indicates that there is an edge from `u` to `s` in the graph `G`.

```
abbreviation parents G s ≡ {u. u →G s}
```

```
function realise :: 'a pset_term ⇒ V where
  x ∈ Var ` pwits b ⟹ realise x = HF {I x}
| x ∈ subterms' b
  ⟹ realise t = HF {realise ` parents (bgraph b) s}
| x ∉ verts G ⟹ realise x = 0
```

Again, we need to ensure that the assumptions of the equations are non-overlapping and that the function terminates. The former is taken care of by automation, leaving us to prove termination. The assumption that `b` is open implies that there are no membership cycles and thus `bgraph b` is acyclic. Furthermore, the graph is finite by definition. Thus, we can use the cardinality of the set of ancestors as a measure that decreases in each recursive call.

Before we prove that the realisation function constitutes a model in subsection 5.2, we will first explain the significance of the pure witnesses.

5.1 Characterisation of the Pure Witnesses

Recall that the pure witnesses of a branch `b` are those witnesses that are not related to other subterms in `last b` by equality. In the context of a well-formed branch, this characterisation can be strengthened to any set term and, in addition, we also get that there is no membership literal where a pure witness is on the right-hand side. Intuitively speaking, the realisation of a pure witness does not depend on the realisation of any other set term.

```
lemma lemma_2:
  assumes wf_branch b and c ∈ pwits b
  shows (Var c =s t) ∉ set b and (t =s Var c) ∉ set b
    and (t ∈s Var c) ∉ set b
```

So why are pure witnesses treated differently? According to the definition of `realise`, the pure witnesses would be evaluated to the empty set `0 :: hf`, were they not treated separately. To see that this is a problem, consider the branch `b = [Var s ≠s Var t, Var t ≠s Var u]` which expands to several open and saturated branches, one of which is

```
  [Var x ≠s Var y, Var x ∈s Var s, Var x ∉s Var t,
                   Var y ∈s Var t, Var y ∉s Var u] @ b
```

for some fresh `x` and `y`. Assigning both `Var x` and `Var y` a value of `0` would contradict the literal `Var x ≠s Var y`. To prevent this, we assign the pure witnesses pairwise different values.

The proof of `lemma_2` is more technical than interesting so we refer the reader to the formalisation.

5.2 Realisation of an Open Branch

Remember that for completeness, we need to show that the realisation function for an open and saturated branch `b` actually constitutes a model for all formulae in the branch. We start by verifying that the realisation function models all literals in the branch; more formally, the following propositions hold:

(1) We have `realise s` $\in$ `realise t` if it holds that `s` $\in_s$ `t` is in `b`.
(2) We have `realise s` $=$ `realise t` if `s` $=_s$ `t` is in `b`.
(3) We have `realise s` $\neq$ `realise t` if `s` $\neq_s$ `t` is in `b`.
(4) We have `realise s` $\notin$ `realise t` if it holds that `s` $\notin_s$ `t` is in `b`.

To illustrate the usefulness of `lemma_2`, we prove Proposition (2). The proofs of all propositions translate well into Isabelle, so we refer to the original paper [9] for the remaining proofs.

Proof (Proof of Proposition (2)). Assume that `s` $=_s$ `t` is in `b`. If there exists a `c` $\in$ `pwits b` where `s = Var c` or `t = Var c`, we arrive at a contradiction due to `lemma_2`. Therefore, both `s` $\in$ `subterms' b` and `t` $\in$ `subterms' b` must hold. Now, assume for contradiction that `realise s` $\neq$ `realise t`. Without loss of generality — the other case is symmetric — we obtain an `e` such that `e` $\in$ `realise s` and `e` $\notin$ `realise t`. Considering that `s` $\in$ `subterms' b` and the definition of `realise`, we obtain a `d` with `e = realise d` and `d` $\rightarrow_{\texttt{bgraph b}}$ `s`. This, in turn, yields that `d` $\in_s$ `s` must be in `b`. Together with the assumption `(s` $=_s$ `t)` $\in$ `set b` and the saturation of `b`, it follows that `d` $\in_s$ `t` must also be in `b`. But then we have `realise d` $\in$ `realise t` $\longleftrightarrow$ `e` $\in$ `realise t` using Proposition (1), which is a contradiction to the assumption `e` $\notin$ `realise t`.

The results on literals can now be lowered to set terms. All of the proofs are straightforward so we refer the reader to the formalisation.

(a) It holds that `realise` $\emptyset$ `= 0`.
(b) Let $\star_s \in \{\sqcup_s, -_s, \sqcap_s\}$. If the term `s` $\star_s$ `t` occurs in `subterms b`, then

$$\texttt{realise (s } \star_s \texttt{ t) = realise s } \star \texttt{ realise t}.$$

(c) If `Single t` $\in$ `subterms b`, then

$$\texttt{realise (Single t) = HF \{realise t\}}.$$

The final step to obtain a proper model is to connect the realisation function to the semantics as defined in section 2. For set terms, we can use the Propositions (a)–(c) to prove the lemma below by induction on `t`.

```
lemma assumes t ∈ subterms b
      shows Iₛₜ (λx. realise (Var x)) t = realise t
```

Lifting the above result to formulae yields the coherence of `b` as the original paper [9] calls it. The proof is a tedious but straightforward induction on the the size of the formulae.

```
lemma coherence:
  assumes φ ∈ set b shows (λx. realise (Var x)) ⊨ φ
```

The coherence property finishes the proof of completeness of the calculus as it gives us a model for every formula in an open and saturated branch.

6 Soundness of the Calculus

A tableau calculus is sound if for any closed tableau, the corresponding formula is unsatisfiable. We prove the following two properties to establish soundness: (1) A closed branch contains an unsatisfiable formula. (2) The expansion rules maintain satisfiability.

We formalise the first property in Isabelle below.

```
lemma bclosed_sound:
  assumes bclosed b shows ∃φ ∈ set b. M ⊭ φ
```

Proof. It is clear that, for any `s`, neither does `M` model $\texttt{s} \in \emptyset$ nor $\texttt{s} \neq_{\texttt{s}} \texttt{s}$. Furthermore, no model can satisfy both ϕ and $\neg\phi$ at the same time. Lastly, a membership cycle is impossible since the membership relation of `hf` is well-founded.

We are left with showing that both linear and branching expansion rules preserve satisfiability. As for the linear rules, a straightforward proof by case analysis on `b' ▷ b` suffices to obtain the lemma below.

```
lemma lexpands_sound:
  assumes b' ▷ b and φ ∈ set b' and ⋀ψ. ψ ∈ set b ⟹ M ⊨ ψ
  shows M ⊨ φ
```

A similar argument would work for the branching rules if it were not for the last rule adding new variables. Those variables need to be assigned specific values; hence, we modify the model as shown in the proof below.

```
lemma bexpands_sound:
  assumes bs' ⊳ b and ⋀ψ. ψ ∈ set b ⟹ M ⊨ ψ
  shows ∃M'. ∃b' ∈ bs'. ∀ψ ∈ set (b' @ b). M' ⊨ ψ
```

Proof. We only consider the case where `bs' ⊳ b` was proved by applying the last branching expansion rule to $\texttt{s} \neq_{\texttt{s}} \texttt{t}$ for some `s` and `t`. We have

$$\texttt{bs'} = \{[\texttt{Var x} \in_{\texttt{s}} \texttt{s},\ \texttt{Var x} \notin_{\texttt{s}} \texttt{t}],\ [\texttt{Var x} \in_{\texttt{s}} \texttt{t},\ \texttt{Var x} \notin_{\texttt{s}} \texttt{s}]\}$$

for some fresh variable `x`. Since $\texttt{s} \neq_{\texttt{s}} \texttt{t}$ is in `b`, we have that $\texttt{I}_{\texttt{st}}\ \texttt{M s} \neq \texttt{I}_{\texttt{st}}\ \texttt{M t}$ because `M` is a model. Without loss of generality this inequality manifests itself through some `y` with $\texttt{y} \in \texttt{I}_{\texttt{st}}\ \texttt{M s}$ and $\texttt{y} \notin \texttt{I}_{\texttt{st}}\ \texttt{M t}$. We update `M` such that it maps `x` to `y` to obtain the assignment `M'`. Note that `M'` is still a model for formulae in `b` because `x` is fresh with respect to `b`. Furthermore, it is also a model for the first branch in `bs'`, which finishes the proof.

7 Total Correctness of the Decision Procedure

We first demonstrate the termination of the procedure for well-formed branches, i.e. every well-formed branch is in the domain of `mlss_proc_branch`. To this end, we derive an upper bound for the number of distinct formulae in a branch whose proof we omit here for brevity (cf. Appendix C). We should point out that this bound is not to be construed as the complexity of the procedure as it may create exponentially many branches in general.

```
lemma card_wf_branch_ub:
  assumes wf_branch b
  shows |set b| ≤ 2 * |subfms (last b)| + 16 * |subterms (last b)|^4
```

Remember that `mlss_proc_branch` only applies a linear expansion rule to a branch if the application results in new formulae. Moreover, the subsumption conditions of the branching expansion rules ensure that each of the newly created branches contain new formulae. Ultimately, we conclude that the procedure must terminate for well-formed branches because the number of formulae increases in each step but is also bounded.

```
lemma assumes wf_branch b shows mlss_proc_branch_dom b
```

The above lemma allows us to utilise the computation induction rule of `mlss_proc_branch` on well-formed branches, which we use to prove soundness and completeness. As both proofs are essentially an application of soundness, respectively completeness, of the calculus, we refer the reader to the formalisation.

```
lemma mlss_proc_branch_complete:
  fixes b :: 'a branch
  assumes wf_branch b and ¬ mlss_proc_branch b
  assumes infinite (UNIV :: 'a set)
  shows ∃M. M ⊨ last b
```

```
lemma mlss_proc_branch_sound:
  assumes wf_branch b and ∀ψ ∈ set b. M ⊨ ψ
  shows ¬ mlss_proc_branch b
```

To finish off the proof of total correctness, note that every singleton branch is trivially well-formed; thus, termination, completeness, and soundness easily transfer to `mlss_proc`.

```
theorem mlss_proc_complete:
  fixes φ :: 'a pset_fm
  assumes ¬ mlss_proc φ and infinite (UNIV :: 'a set)
  shows ∃M. M ⊨ φ
```

```
theorem mlss_proc_sound:
  assumes M ⊨ φ shows ¬ mlss_proc φ
```

8 Dealing with Urelements

In the introduction, we stated the goal of integrating `mlss_proc` as a tactic into Isabelle. For this to work, we need to map every branch expansion rule to a corresponding theorem in Isabelle/HOL. This is straightforward for all expansion rules except for the last branching expansion rule. To illustrate, suppose that we are to disprove a statement of the form

```
s ≠ (t :: 'a) ∧ s ∈ (A :: 'a set) ∪ B ∧ ...
```

in Isabelle/HOL. By way of reification, we convert this to a formula of the shape $\texttt{s'} \neq_s \texttt{t'} \wedge \texttt{s'} \in_s \texttt{A'} \sqcup_s \texttt{B'} \wedge \ldots$ in our set syntax for some `s'`, `t'`, `A'`, and `B'`. When we apply the decision procedure to this formula, we might get back a tableau proof that contains an application of the last branching rule to $(\texttt{s'} \neq_s \texttt{t'}) \in \texttt{set b}$. This results in two branches, one of which is $[\texttt{Var x} \in_s \texttt{s'}, \texttt{Var x} \notin_s \texttt{t'}]\ \texttt{@ b}$; however, there is no matching rule in Isabelle/HOL since `s` and `t` are not sets.

To deal with this problem, we formalise a light-weight type system as displayed in Figure 1. The type of a set term in this system is just a natural number which we call level. Intuitively speaking, the level indicates that the corresponding term `t` in Isabelle/HOL has type

$$\texttt{'a}\ \underbrace{\texttt{set} \ldots \texttt{set}}_{\texttt{l}\ \text{times}}$$

for some `'a`. Note that the constructor $\emptyset$ now receives an additional argument that indicates the level of each instance of $\emptyset$.

Moreover, the typing judgement extends to set atoms by matching up the levels of its component set terms.

Ultimately, we define $\Gamma \vdash \phi \equiv \forall \texttt{a} \in \texttt{atoms}\ \phi.\ \Gamma \vdash \texttt{a}$ in order to type formulae.

We can now define the urelements with respect to a formula. An urelement is a set term whose corresponding type in Isabelle/HOL might not be a set.

```
definition urelem :: 'a pset_fm ⇒ 'a pset_term ⇒ bool where
  urelem φ t ≡ ∃Γ. Γ ⊢ φ ∧ Γ ⊢ t : 0
```

$$\frac{}{\Gamma \vdash \emptyset\ \texttt{n} : \texttt{Suc n}} \qquad \frac{}{\Gamma \vdash \texttt{Var x} : \Gamma\ \texttt{x}} \qquad \frac{\Gamma \vdash \texttt{t} : \texttt{l}}{\Gamma \vdash \texttt{Single t} : \texttt{Suc l}}$$

$$\frac{\star_s \in \{\sqcup_s, \sqcap_s, -_s\} \quad \Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{l} \quad \texttt{l} \neq 0}{\Gamma \vdash \texttt{s} \star_s \texttt{t} : \texttt{l}}$$

$$\frac{\Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{l}}{\Gamma \vdash \texttt{s} =_s \texttt{t}} \qquad \frac{\Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{Suc l}}{\Gamma \vdash \texttt{s} \in_s \texttt{t}}$$

Figure 1. The type system for set terms and atoms.

Using this definition, we make two changes to the specification of the calculus: (1) First and foremost, we require that neither `s` nor `t` is an urelement in the precondition of the last branching expansion rule. (2) As mentioned above, we add an argument to the $\emptyset$ constructor. This argument is only used for the typing judgement; it has no impact on the semantics.

Soundness, of course, is not affected by these changes but we have to make a few amendments to maintain completeness: (1) The first equation of `realise` now also needs to account for the urelements. In particular, it needs to ensure that urelements receive pairwise different values unless they are related through equality atoms. This has no effect on pure witnesses since they can not be related through equality atoms due to `lemma_2`. (2) We need to adjust the completeness proof in those places where it directly refers to the definition of `realise` to account for the case where a given term is an urelement. (3) The completeness theorem receives the additional assumption that $\Gamma \vdash \phi$ holds for the initial formula ϕ. (4) For the completeness proof, we need to show that the typing judgement is invariant under branch expansion.

The modifications above ensure that the proof can be replayed through Isabelle/HOL. To actually use the calculus, we need to determine the urelements of the initial formula ϕ, though. In other words, we need to implement an inference algorithm for our light-weight type system. The algorithm is, in essence, a simplified version of Hindley-Milney type inference so it has the same two phases: it generates constraints using syntax directed rules and then passes them to a constraint solver.

Since we are only interested in the level of a term, we can encode all constraints into the theory consisting of 0, the successor function `S`, and equality (but no disequality). Note that constraints of the form $\texttt{l} \neq \texttt{0}$ can be replaced by $\texttt{l} = \texttt{S i}$ with `i` being a fresh variable. A solver for this theory is straightforward to implement and verify; nevertheless, we have to be careful that it computes the minimum assignment Γ from variables to levels that fulfils the constraints. This is to guarantee that a set term `t` is not an urelement **iff** $\Gamma\ \texttt{t} > \texttt{0}$. Conversely, all terms `s` with $\Gamma\ \texttt{s} = \texttt{0}$ are urelements.

9 Conclusion and Future Work

By closely following a paper by Cantone and Zarba [9], we developed a formalisation of a tableau calculus for a quantifier-free fragment of set theory called **MLSS**. The formalisation includes an abstract description of a decision procedure that builds on the calculus. To make the decision procedure compatible with Isabelle/HOL, we extended the calculus with a light-weight type system while maintaining completeness. We also refined the abstract specification to an executable specification from which code can be generated.

In future work, we plan to implement an efficient executable specification in the style of a worklist algorithm. This specification should also generate certificates that can be replayed through Isabelle's inference kernel in order to facilitate the integration of the procedure into Isabelle.

Bibliography

[1] Beckert, B., Hartmer, U.: A tableau calculus for quantifier-free set theoretic formulae. Automated Reasoning with Analytic Tableaux and Related Methods p. 93–107 (1998), https://doi.org/10.1007/3-540-69778-0_16

[2] Bentzen, B.: A Henkin-style completeness proof for the modal logic S5. In: Baroni, P., Benzmüller, C., Wáng, Y.N. (eds.) Logic and Argumentation - 4th International Conference, Lecture Notes in Computer Science, vol. 13040, pp. 459–467, Springer (2021), https://doi.org/10.1007/978-3-030-89391-0_25

[3] Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. Journal of Automated Reasoning **58**(1), 149–179 (2017), https://doi.org/10.1007/s10817-016-9391-3

[4] Cantone, D.: A fast saturation strategy for set-theoretic tableaux. In: Galmiche, D. (ed.) Automated Reasoning with Analytic Tableaux and Related Methods, pp. 122–137, Springer (1997), https://doi.org/10.1007/BFb0027409

[5] Cantone, D., Longo, C., Asmundo, M.N.: A decision procedure for a two-sorted extension of multi-level syllogistic with the cartesian product and some map constructs. In: Faber, W., Leone, N. (eds.) Italian Conference on Computational Logic, CEUR Workshop Proceedings, vol. 598, CEUR-WS.org (2010), URL http://ceur-ws.org/Vol-598/paper11.pdf

[6] Cantone, D., Omodeo, E.G., Policriti, A.: The automation of syllogistic. ii. optimization and complexity issues. Journal of Automated Reasoning **6**(2), 173–187 (1990), ISSN 0168-7433, https://doi.org/10.1007/BF00245817

[7] Cantone, D., Omodeo, E.G., Policriti, A.: Set Theory for Computing - From Decision Procedures to Declarative Programming with Sets. Monographs in Computer Science, Springer (2001), https://doi.org/10.1007/978-1-4757-3452-2

[8] Cantone, D., Schwartz, J.T., Zarba, C.G.: A decision procedure for a sublanguage of set theory involving monotone, additive, and multiplicative functions. Electronic Notes in Theoretical Computer Science **86**(1), 49–60 (2003), https://doi.org/10.1016/S1571-0661(04)80652-2, international Workshop on First-Order Theorem Proving

[9] Cantone, D., Zarba, C.G.: A new fast tableau-based decision procedure for an unquantified fragment of set theory. In: Caferra, R., Salzer, G. (eds.) Automated Deduction in Classical and Non-Classical Logics, Selected Papers, Lecture Notes in Computer Science, vol. 1761, pp. 126–136, Springer (1998), https://doi.org/10.1007/3-540-46508-1_8

[10] Cantone, D., Zarba, C.G.: A tableau-based decision procedure for a fragment of set theory involving a restricted form of quantification. In: Murray, N.V. (ed.) Automated Reasoning with Analytic Tableaux and Related Methods, Lecture Notes in Computer Science, vol. 1617, pp. 97–112, Springer (1999), https://doi.org/10.1007/3-540-48754-9_12

[11] Chaieb, A.: Verifying mixed real-integer quantifier elimination. In: Furbach, U., Shankar, N. (eds.) International Joint Conference on Automated Reasoning, Lecture Notes in Computer Science, vol. 4130, pp. 528–540, Springer (2006), https://doi.org/10.1007/11814771_43

[12] Chaieb, A., Nipkow, T.: Verifying and reflecting quantifier elimination for presburger arithmetic. In: Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, Lecture Notes in Computer Science, vol. 3835, pp. 367–380, Springer (2005), https://doi.org/10.1007/11591191_26

[13] Doczkal, C., Smolka, G.: Completeness and decidability results for ctl in coq. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving, pp. 226–241, Springer International Publishing (2014), https://doi.org/10.1007/978-3-319-08970-6_15

[14] Ferro, A., Omodeo, E.G., Schwartz, J.T.: Decision procedures for elementary sublanguages of set theory. i. multi-level syllogistic and some extensions. Communications on Pure and Applied Mathematics **33**(5), 599–608 (1980), https://doi.org/10.1002/cpa.3160330503

[15] Fitting, M.: Semantic Tableaux and Resolution. Springer New York, New York, NY (1996), https://doi.org/10.1007/978-1-4612-2360-3_3

[16] From, A.H.: Formalizing a Seligman-style tableau system for hybrid logic. Archive of Formal Proofs (December 2019), ISSN 2150-914x, https://isa-afp.org/entries/Hybrid_Logic.html, Formal proof development

[17] From, A.H.: Synthetic completeness. Archive of Formal Proofs (January 2023), ISSN 2150-914x, https://isa-afp.org/entries/Synthetic_Completeness.html, Formal proof development

[18] From, A.H., Blackburn, P., Villadsen, J.: Formalizing a Seligman-style tableau system for hybrid logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning, pp. 474–481, Springer International Publishing, Cham (2020), https://doi.org/10.1007/978-3-030-51074-9_27

[19] From, A.H., Schlichtkrull, A., Villadsen, J.: A sequent calculus for first-order logic formalized in Isabelle/HOL. In: Monica, S., Bergenti, F. (eds.) Proceedings of the 36th Italian Conference on Computational Logic, CEUR Workshop Proceedings, vol. 3002, pp. 107–121, CEUR-WS.org (2021), URL http://ceur-ws.org/Vol-3002/paper7.pdf

[20] Nipkow, T.: Linear quantifier elimination. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, pp. 18–33, Springer (2008), https://doi.org/10.1007/978-3-540-71070-7_3

[21] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[22] Noschinski, L.: Graph theory. Archive of Formal Proofs (April 2013), ISSN 2150-914x, https://isa-afp.org/entries/Graph_Theory.html, Formal proof development

[23] Paulson, L.C.: The hereditarily finite sets. Archive of Formal Proofs (November 2013), ISSN 2150-914x, https://isa-afp.org/entries/HereditarilyFinite.html, Formal proof development

[24] Paulson, L.C.: Zermelo fraenkel set theory in higher-order logic. Archive of Formal Proofs (October 2019), ISSN 2150-914x, `https://isa-afp.org/entries/ZFC_in_HOL.html`, Formal proof development

[25] Stevens, L.: Formalisation of a decision procedure for MLSS (2023), URL `https://github.com/lukasstevens/set-decision-procedure/tree/CADE`, Formal proof development

[26] Stevens, L., Nipkow, T.: A verified decision procedure for orders in Isabelle/HOL. In: Hou, Z., Ganesh, V. (eds.) Automated Technology for Verification and Analysis, pp. 127–143, Springer International Publishing (2021), `https://doi.org/10.1007/978-3-030-88885-5_9`

Appendix

A Subterms and Subformulas

We introduce the function `subterms :: 'b ⇒ 'a pset_term` that is overloaded in its argument. We first overload the function for set terms.

```
fun subterms :: 'a pset_term ⇒ 'a pset_term set where
  subterms ∅ = {∅}
| subterms (Var x) = {Var x}
| subterms (s ⊔s t) = {s ⊔s t} ∪ subterms s ∪ subterms t
| subterms (s ⊓s t) = {s ⊓s t} ∪ subterms s ∪ subterms t
| subterms (s −s t) = {s −s t} ∪ subterms s ∪ subterms t
| subterms (Single t) = {Single t} ∪ subterms t
```

The subterms of an atom are then the subterms of its arguments.

```
fun subterms :: 'a pset_atom ⇒ 'a pset_term set where
  subterms (s ∈s t) = subterms s ∪ subterms t
| subterms (s =s t) = subterms s ∪ subterms t
```

Finally, we lift this to formulae by using `atoms` again.

```
definition subterms :: 'a pset_fm ⇒ 'a pset_term set where
  subterms φ ≡ ⋃(subterms ` atoms φ)
```

We also consider the subformulae of a formula and define a function that computes them.

```
fun subfms :: 'a fm ⇒ 'a fm set where
  subfms (A a) = {A a}
| subfms (p ∧ q) = {p ∧ q} ∪ subfms p ∪ subfms q
| subfms (p ∨ q) = {p ∨ q} ∪ subfms p ∪ subfms q
| subfms (¬ q) = {¬ q} ∪ subfms q
```

Finally, we lift the functions `vars` and `subterms` to branches.

```
definition vars :: 'a branch ⇒ 'a set
  vars b ≡ ⋃(vars ` set b)

definition subterms :: 'a branch ⇒ 'a pset_term set where
  subterms b ≡ ⋃(subterms ` set b)
```

B Definition of the Interpretation Functions

We define the interpretation function $\mathtt{I}_{\mathtt{st}}$ for set atoms.

```
fun Ist :: ('a ⇒ hf) ⇒ 'a pset_term ⇒ hf where
  Ist M ∅ = 0
| Ist M (Var x) = M x
| Ist M (Single s) = HF {Ist M s}
| Ist M (s ⊔s t) = Ist M s ⊔ Ist M t
| Ist M (s ⊓s t) = Ist M s ⊓ Ist M t
| Ist M (s −s t) = Ist M s − Ist M t
```

The interpretation function $\mathtt{I}_{\mathtt{sa}}$ for set atoms is straightforward as well.

```
fun Isa :: ('a ⇒ hf) ⇒ 'a pset_atom ⇒ bool where
  Isa M (s ∈s t) ⟷ Ist M s ∈ Ist M t
| Isa M (s =s t) ⟷ Ist M s = Ist M t
```

C Upper-Bounding the Cardinality of a Branch

The argument below argues about the *literals* of a branch, which are defined as follows.

```
fun is_literal :: 'a fm ⇒ bool where
  is_literal (A _) = True | is_literal (¬ (A _)) = True
| is_literal _ = False
```

We still need to give a termination proof to ascertain total correctness. The core of our argument is that the number of distinct formulae that can be derived with the expansion rules is finite. In particular, we claim that the following upper bound holds.

$$\begin{aligned}
& |\mathtt{set\ b}| \\
= {} & |\{\psi \in \mathtt{set\ b.}\ \neg\ \mathtt{is\_literal}\ \psi\}| + \\
& |\{\psi \in \mathtt{set\ b.\ is\_literal}\ \psi\}| \\
\le {} & 2 * |\mathtt{subfms\ (last\ b)}| + |\{\psi \in \mathtt{set\ b.\ is\_literal}\ \psi\}| \\
\le {} & 2 * |\mathtt{subfms\ (last\ b)}| + 16 * |\mathtt{subterms\ (last\ b)}|^4
\end{aligned}$$

We treat both inequalities separately, starting with the first one whose justification hinges on an invariant: every new composite subformula that is introduced by an expansion rule is the negation of a subformula that already occurs in `last b`. Written as a formula, we have

$$\begin{aligned}
& \{\psi \in \mathtt{set\ b.}\ \neg\ \mathtt{is\_literal}\ \psi\} \\
\subseteq {} & \mathtt{subfms\ (last\ b)} \cup \neg\ \text{‘}\ \mathtt{subfms\ (last\ b).}
\end{aligned}$$

Confirming that this is an invariant is an easy exercise of going through each of the propositional rules in Table 1 and the first two branching rules in Table 2. Thus, we have

```
  |{ψ ∈ set b. ¬ is_literal ψ}|
≤ |subfms (last b)| + |¬ ‘ subfms (last b)|
≤ 2 * |subfms (last b)|
```

which is the first inequality we asserted.

Concerning the second inequality, first observe that each literal contains an atom, implying that the set of literals is at most twice as large as the set of atoms in a branch. In addition, an atom is formed by combining two subterms occurring in the branch with either $\in_{\mathtt{s}}$ or $=_{\mathtt{s}}$; hence, we have

```
  |{ψ ∈ set b. is_literal ψ}|
≤ 2 * |⋃(atoms ‘ set b)| ≤ 2 * 2 * |subterms b|^2.
```

Since no expansion rule introduces new subterms, we know the cardinality of the subterms in a branch `b` is limited by the subterms of `last b` plus the number of witnesses added to the branch. Note that the last branching rule requires `s` $\neq_{\mathtt{s}}$ `t` to be in `b` for some `s,t ∈ subterms (last b)`. The subsumption condition then prohibits that the rule can be applied again for the combination of `s` and `t`, thus yielding an upper bound of `|subterms (last b)|`^{2} for the witnesses. This allows us to finish the calculation that justifies the second inequality:

```
  4 * |subterms b|^2
≤ 4 * |subterms (last b)| + |wits b|^2
≤ 4 * |subterms (last b)| + (|subterms (last b)|^2)^2
≤ 4 * 4 * |subterms (last b)|^4.
```

All in all, we get the following lemma.

```
lemma card_wf_branch_ub:
  assumes wf_branch b
  shows |set b|
      ≤ 2 * |subfms (last b)| + 16 * |subterms (last b)|^4
```

D Linear Expansion Rules

Table 3. Linear expansion rules.

Propositional Rules		
$p \wedge q$	$\Longrightarrow$	$p,\ q$
$\neg(p \vee q)$	$\Longrightarrow$	$\neg p,\ \neg q$
$p \vee q,\ \neg p$	$\Longrightarrow$	q
$p \vee q,\ \neg q$	$\Longrightarrow$	p
$\neg(p \wedge q),\ p$	$\Longrightarrow$	$\neg q$
$\neg(p \wedge q),\ q$	$\Longrightarrow$	$\neg p$
$\neg(\neg p)$	$\Longrightarrow$	p

Rules for $\sqcup_s$		
$s \notin_s t_1 \sqcup_s t_2$	$\Longrightarrow$	$s \notin_s t_1,$ $s \notin_s t_2$
$s \in_s t_1$	$\Longrightarrow$	$s \in_s t_1 \sqcup_s t_2$
$s \in t_2$	$\Longrightarrow$	$s \in_s t_1 \sqcup_s t_2$
$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_1$	$\Longrightarrow$	$s \in_s t_2$
$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_2$	$\Longrightarrow$	$s \in_s t_1$
$s \notin_s t_1,\ s \notin_s t_2$	$\Longrightarrow$	$s \notin_s t_1 \sqcup_s t_2$

Rules for $\sqcap_s$		
$s \in_s t_1 \sqcap_s t_2$	$\Longrightarrow$	$s \in_s t_1,$ $s \in_s t_2$
$s \notin_s t_1$	$\Longrightarrow$	$s \notin_s t_1 \sqcap_s t_2$
$s \notin_s t_2$	$\Longrightarrow$	$s \notin_s t_1 \sqcap_s t_2$
$s \notin_s t_1 \sqcap_s t_2,$ $s \in_s t_1$	$\Longrightarrow$	$s \notin_s t_2$
$s \notin_s t_1 \sqcap_s t_2$ $s \in_s t_2$	$\Longrightarrow$	$s \notin_s t_1$
$s \in_s t_1,\ s \in_s t_2$	$\Longrightarrow$	$s \in_s t_1 \sqcap_s t_2$

Rules for $-_s$		
$s \in_s t_1 -_s t_2$	$\Longrightarrow$	$s \in_s t_1,$ $s \notin_s t_2$
$s \notin_s t_1$	$\Longrightarrow$	$s \notin_s t_1 -_s t_2$
$s \in_s t_2$	$\Longrightarrow$	$s \notin_s t_1 -_s t_2$
$s \notin_s t_1 -_s t_2,$ $s \in_s t_1$	$\Longrightarrow$	$s \in_s t_2$
$s \notin_s t_1 -_s t_2,$ $s \notin_s t_2$	$\Longrightarrow$	$s \notin_s t_1$
$s \in_s t_1,\ s \notin_s t_2$	$\Longrightarrow$	$s \in_s t_1 -_s t_2$

Rules for `Single`		
	$\Longrightarrow$	$s \in_s \texttt{Single}\ s$
$s \in_s \texttt{Single}\ t$	$\Longrightarrow$	$s =_s t$
$s \notin_s \texttt{Single}\ t$	$\Longrightarrow$	$s \neq_s t$

Rules for $=_s$		
$t_1 =_s t_2,\ l$	$\Longrightarrow$	$l\{t_2/t_1\}$
$t_1 =_s t_2,\ l$	$\Longrightarrow$	$l\{t_1/t_2\}$
$s_1 \in_s t,\ s_2 \notin_s t$	$\Longrightarrow$	$s_1 \neq_s s_2$

E Branching Expansion Rules

Table 4. Branching expansion rules. We write ϕ for `last b` here.

Rule	Precondition	Subsumption condition
$\dfrac{}{\mathtt{p} \mid \neg\,\mathtt{p}}$	$\mathtt{p} \vee \mathtt{q} \in \mathtt{set\ b}$	$\mathtt{p} \in \mathtt{set\ b} \vee$ $\neg\,\mathtt{p} \in \mathtt{set\ b}$
$\dfrac{}{\neg\,\mathtt{p} \mid \mathtt{p}}$	$\neg\,(\mathtt{p} \wedge \mathtt{q}) \in \mathtt{set\ b}$	$\mathtt{p} \in \mathtt{set\ b} \vee$ $\neg\,\mathtt{p} \in \mathtt{set\ b}$
$\dfrac{}{\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1 \mid \mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_1}$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1 \sqcup_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$ $\mathtt{t}_1 \sqcup_{\mathtt{s}} \mathtt{t}_2 \in \mathtt{subterms}\ \phi$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$ $\vee\ (\mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$
$\dfrac{}{\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_2 \mid \mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_2}$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$ $(\mathtt{t}_1 \sqcap_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{subterms}\ \phi$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$ $\vee\ (\mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$
$\dfrac{}{\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_2 \mid \mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_2}$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$ $(\mathtt{t}_1 -_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{subterms}\ \phi$	$(\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$ $\vee\ (\mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$
$\dfrac{}{\begin{array}{l\|l} \mathtt{Var\ x} \in_{\mathtt{s}} \mathtt{t}_1 & \mathtt{Var\ x} \notin_{\mathtt{s}} \mathtt{t}_1 \\ \mathtt{Var\ x} \notin_{\mathtt{s}} \mathtt{t}_2 & \mathtt{Var\ x} \in_{\mathtt{s}} \mathtt{t}_2 \end{array}}$	$(\mathtt{t}_1 \neq_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$ $\mathtt{t}_1 \in \mathtt{subterms}\ \phi$ $\mathtt{t}_2 \in \mathtt{subterms}\ \phi$ $\mathtt{x} \notin \mathtt{vars\ b}$	$\exists \mathtt{s}.\ (\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$ $\wedge\ (\mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$ $\vee$ $\exists \mathtt{s}.\ (\mathtt{s} \notin_{\mathtt{s}} \mathtt{t}_1) \in \mathtt{set\ b}$ $\wedge\ (\mathtt{s} \in_{\mathtt{s}} \mathtt{t}_2) \in \mathtt{set\ b}$