A Verified Prover Based on Ordered Resolution
Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel

To cite this version:
Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel. A Verified Prover Based on Ordered Resolution. CPP 2019 - The 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2019, Cascais, Portugal. CPP 2019 - The 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. <10.1145/3293880.3294100>. <hal-01937141>
A Verified Prover Based on Ordered Resolution

Anders Schlichtkrull  
DTU Compute  
Technical University of Denmark  
Kongens Lyngby, Denmark  
andschl@dtu.dk

Jasmin Christian Blanchette  
Department of Computer Science  
Vrije Universiteit Amsterdam  
Amsterdam, the Netherlands  
j.c.blanchette@vu.nl

Dmitriy Traytel  
Department of Computer Science  
ETH Zürich  
Zürich, Switzerland  
traytel@inf.ethz.ch

Abstract

The superposition calculus, which underlies first-order theorem provers such as E, SPASS, and Vampire, combines ordered resolution and equality reasoning. As a step towards verifying modern provers, we specify, using Isabelle/HOL, a purely functional first-order ordered resolution prover and establish its soundness and refutational completeness. Methodologically, we apply stepwise refinement to obtain, from an abstract nondeterministic specification, a verified deterministic program, written in a subset of Isabelle/HOL from which we extract purely functional Standard ML code that constitutes a semidecision procedure for first-order logic.

CCS Concepts  → Theory of computation  → Logic and verification;  Automated reasoning;

Keywords  automatic theorem provers, proof assistants, first-order logic, stepwise refinement

ACM Reference Format:

1 Introduction

Automatic theorem provers based on superposition, such as E [42], SPASS [33], and Vampire [21], are often employed as backends in proof assistants and program verification tools [7, 20, 32]. Superposition is a highly successful calculus for first-order logic with equality, which generalizes both resolution [2] and ordered resolution [1].

Resolution operates on sets of clauses. A clause is an n-ary disjunction of literals $L_1 \lor \cdots \lor L_n$ whose variables are interpreted universally. Each literal is either an atom $A$ or its negation $\neg A$. An atom is a symbol applied to a tuple of terms—e.g., prime($n$). The empty clause is denoted by $\bot$.

Resolution works by refutation: Conceptually, the calculus proves a conjecture $\forall x. C$ from a set of axioms $D$ by deriving $\bot$ from $D \cup \{ \exists x. \neg C \}$, indicating its unsatisfiability. As an optimization, it uses a redundancy criterion to discard tautologies, subsumed clauses, and other unnecessary clauses; for example, $p(x) \lor q(x)$ and $p(5)$ are both subsumed by $p(x)$. Compared with plain resolution, ordered resolution relies on an order on the atoms to further prune the search space.

Modern superposition provers are highly optimized programs that rely on sophisticated calculi, with a rich metatheory. In this paper, we propose to verify, using Isabelle/HOL [30], a purely functional prover based on ordered resolution. Although our primary interest is in metatheory per se, there are of course applications for verified provers [50].

The verification relies on stepwise refinement [55]. Four layers are connected by three refinement steps.

Our starting point, layer 1 (Section 3), is an abstract Prolog-style nondeterministic resolution prover in a highly general form, as presented by Bachmair and Ganzinger [2] and as formalized in our earlier work [39, 40]. It operates on possibly infinite sets of clauses. Its soundness and refutational completeness are inherited by the other layers.

Layer 2 (Section 4) operates on finite multisets of clauses and introduces a priority queue to ensure that inferences are performed in a fair manner, guaranteeing completeness: Given a valid conjecture, the prover will eventually derive $\bot$.

Layer 3 (Section 5) is a deterministic program that works on finite lists, committing to a strategy for assigning priorities to clauses. However, it is not fully executable: It abstracts over operations on atoms and employs logical specifications instead of executable functions for auxiliary notions.

Finally, layer 4 (Section 6) is a fully executable program. It provides a concrete datatype for atoms and executable definitions for all auxiliary notions, including unifiers, clause subsumption, and the order on atoms.

From layer 4, we can extract Standard ML code by invoking Isabelle’s code generator [11]. The resulting prover constitutes a proof of concept: It uses an efficient calculus (layer 1) and a reasonable strategy to ensure fairness (layers 2 and 3), but depends on inefficient list-based data structures. Further refinement steps will be required to obtain a prover that is competitive with the state of the art.
The refinement steps connect vastly different levels of abstraction. The most abstract level is occupied by an infinitary logical calculus and the semantics of first-order logic. Soundness and completeness relate these two notions. At the functional programming level, soundness amounts to a safety property: Whenever the program terminates normally, its outcome is correct, whether it is a proof or a finite saturation witnessing unprovability. Correspondingly, refutational completeness is a liveness property: If the conjecture is valid, the program will always terminate normally. We find that, far from being academic exercises, Bachmair and Ganzinger’s framework [2] and its formalization [39, 40] adequately capture the metatheory of actual provers.

To our knowledge, our program is the first verified prover for first-order logic implementing an optimized calculus. It is also the first example of the application of refinement in a first-order context. This methodology has been used to verify SAT solvers [6, 29], which decide the satisfiability of propositional formulas, but first-order logic is undecidable—sound and complete provers are guaranteed to terminate only for unsatisfiable (i.e., provable) clause sets. This complicates the transfer of completeness results across refinement layers.

The present work is part of the IsaFoL (Isabelle Formalization of Logic) project, which aims at developing a library of results about logic and automated reasoning [3]. The IsaFoL files are available in the Archive of Formal Proofs [38, 39] and in the IsaFoL repository. The parts specific to the functional and final layer, the abstract framework is instantiated with a safety property: Whenever the program terminates normally, its outcome is correct, whether it is a proof or a finite satisfaction witnessing unprovability. The truth value of ground (i.e., variable-free) atoms is given by a Herbrand interpretation: a set I, of type ’a set, of all true ground atoms. The “models” predicate ⪯ is defined as I ⪯ A ⇔ A ∈ I. This definition is lifted to literals, clauses, and sets of clauses in the usual way. A set of clauses D is satisfiable if there exists an interpretation I such that I ⪯ D.

Resolution depends on a notion of substitution of and most general unifier (MGU). These auxiliary concepts are provided by a third-party library, IsaFoR (Isabelle Formalization of Rewriting) [51]. To reduce our dependency on external libraries, we hide them behind abstract locales parameterized by a type of atoms ’a and a type of substitutions ’s.

We start by defining a locale substitution_ops that declares application (·), identity (id), and composition (◦):

```
locale substitution_ops = fixes id :: ’a and o :: ’s ⇒ ’s and · :: ’a ⇒ ’s ⇒ ’a
```

Within the locale’s scope, we introduce a number of derived concepts. Ground atoms are defined as those atoms that are left unchanged by substitutions: is_ground A ⇔ ∀σ. A = A · σ. A ground substitution is a substitution whose application always results in ground atoms. Nonstrict and strict generalization are defined as

```
generalizes A B ⇔ ∃σ. A · σ = B
strictly_generalizes A B ⇔ generalizes A B
∧ ¬ generalizes B A
```

The operators on atoms are lifted to literals, clauses, and sets of clauses. The grounding of a clause is defined as

```
grounding_of C = {C · σ | is_ground σ}
```

The operator is lifted to sets of clauses in the obvious way. Clause subsumption is defined as

```
subsumes C D ⇔ ∃σ. C · σ ⊆ D
```

with strictly_subsumes as its strict counterpart.

The next locale, substitution, characterizes the operations defined by substitution_ops. A separate locale is necessary because we cannot interleave assumptions and definitions in a single locale. In addition, substitution fixes a function for renaming clauses apart (so that they do not share any variables) and a function that, given a list of atoms, constructs an atom with these as subterms:

```
locale substitution = substitution_ops + fixes renamings_apart :: ’a clause list ⇒ ’s list and atm_of_atms :: ’a list ⇒ ’a
assumes A · id = A A · (σ ◦ τ) = (A · σ) · τ (∀A. A · σ = A · τ) ⇒ σ = τ is_ground (C · σ) ⇒ ∃r. is_ground r ∧ C · r = C · σ
```

1https://bitbucket.org/isafol/isafol/wiki/Home
wf strictly_generalizes
|renamings_apart Cs| = |Cs|
ρ ∈ renamings_apart Cs ⇒ is_renamings ρ
var_disjoint (Cs · renamings_apart Cs)
atom_of_atms As · σ = atom_of_atms Bs ⇐⇒
map (λA. A · σ) As = Bs

The above definition is presented to give a flavor of our development. We refer to the Isabelle files for the exact definitions. Inside the locale, we prove further properties of the substitution_ops operations. Notably, we prove well-foundedness of the strictly_subsumes predicate based on the well-foundedness of strictly_generalizes, which is stated as an assumption. The atm_of_atms operation is used to encode a clause as a single atom in this well-foundedness proof.

Finally, a third locale, mgu, extends substitution by fixing a function mgu :: "a set set ⇒ 's option" that computes an MGU σ given a set of unification constraints.

3 Bachmair and Ganzinger’s Prover

Our earlier formalization [39, 40] of a nondeterministic ordered resolution prover presented by Bachmair and Ganzinger [2] forms layer 1 of our refinement. In this paper, we restrict our focus to binary resolution, which can be implemented efficiently and forms the basis of modern provers.

The ordered resolution calculus is parameterized by a total order > ("larger than") on ground atoms. For first-order logic, the order > is extended to an order > on nonground atoms so that B > A if and only if for all ground substitutions σ, we have B · σ > A · σ. The calculus consists of the single rule

$$
C ∨ A₁ ∨ ⋯ ∨ A_k ∨ ¬A ∨ D
$$

where σ is the (canonical) MGU that solves the unification problem A₁ ≡ ⋯ ≡ Aₖ ≡ A. In addition, each Aᵢ · σ must be strictly >-maximal with respect to the atoms in C · σ (meaning that Aᵢ is not ≤ any atom in C · σ), and A · σ is >-maximal with respect to the atoms in D · σ. To achieve completeness, the rule must be adapted slightly to rename apart the variables occurring in different premises.

A set of clauses D is saturated if any conclusion from premises in D is already in D. The ordered resolution calculus is refutation complete, meaning that any unsatisfiable saturated set of clauses necessarily contains ⊥.

Resolution provers start with a finite set of initial clauses—the input problem—and successively add conclusions from premises in the set. If the inference rule is applied in a fair fashion, the set reaches saturation at the limit; if the set is unsatisfiable, this means ⊥ is eventually derived.

Crucially, not only do efficient provers add clauses to their working set, they also remove clauses that are deemed redundant. This requires a refined notion of saturation. We call a set of clauses D saturated up to redundancy, written saturated_up_to D, if any inference from nonredundant clauses in D yields a redundant conclusion.

Bachmair and Ganzinger’s nondeterministic first-order prover, called RP, captures the “dynamic” aspects of saturation. RP is defined as an inductive predicate ⊢ on states, which are triples S = (N, P, O) of new clauses N, processed clauses P, and old clauses O. Initially, N is the input problem, and P ∪ O is empty. Clauses can be removed if they are tautological or subsumed or after subsumption resolution has been applied. When all clauses in N have been processed (either removed entirely or moved to P), a clause C from P can be chosen for inference computation: C is then moved from P to O, and all its conclusions with premises from the other old clauses form the new N. Formally:

inductive ⊢ :: 'a state ⇒ 'a state ⇒ bool where
Neg A ∈ C ∧ Pos A ∈ C ⇒
(N ∪ {C}, P, O) ⊢ (N ∪ {C}, P, O)
| D ∈ P ∪ O ∧ subsumes D C ⇒
(N ∪ {C}, P, O) ⊢_{−ω} (N ∪ {C}, P, O)
| D ∈ N ∩ strictly_subsumes D C ⇒
(N, P ∪ {C}, O) ⊢_{−ω} (N, P ∪ {C}, O)
| D ∈ N ∩ strictly_subsumes D C ⇒
(N, P, O ∪ {C}) ⊢_{−ω} (N, P, O ∪ {C})
| D ∈ P ∪ O ∧ reduces D C L ⇒
(N, P ∪ O ∪ {C ∪ {l}}), O) ⊢_{−ω} (N, P ∪ O ∪ {C ∪ {l}}, O)
| (N ∪ {C}, P, O) ⊢_{−9} (N, P ∪ {C}, O)
| (0, P ∪ {C}, O) ⊢_{−9} (concl_of’ infers_between O C, P, O ∪ {C})

Subscripts on ⊢ identify the rules. The notation f ∘ X stands for the image of X under f, infers_between O C calculates all the ordered resolution inferences whose premises are a subset of O ∪ C that contains C, and reduces D C L is defined as ∃D’ L’ ∨ C. D = D’ ∪ {l} ∧ L = L’ ∨ D’ ∨ C.

The following derivation shows that RP can diverge even on unsatisfiable clause sets:

$$
\exists(p'(a, a), p'(x, x), ∼p'(f(x), y) ∨ p(x, y)), \{p', 0\}, \{ \{∼p(a, a), p(x, x), ∼p(f(x), y) ∨ p(x, y)\}, 0\} \sim_{−9} (0, \{∼p(a, a), p(x, x), ∼p(f(x), y) ∨ p(x, y)\}) \sim_{−9} (0, \{∼p(a, a), p(x, f(x))\}, ∼p(f(x), y) ∨ p(x, y), p(x, x)) \sim_{−9} (0, \{∼p(a, a), p(x, f(x))\}, ∼p(f(x), y) ∨ p(x, y), p(x, f(x))) \sim_{−9} (0, \{∼p(a, a), p(x, f(x))\}, ∼p(f(x), y) ∨ p(x, y), p(x, f(x))) \sim_{−9} \cdots
$$

We can leave ∼p(a, a) in P forever and always generate more clauses of the form p(x, fⁱ(x)), for increasing values of i. This emphasizes the importance of a fair strategy for selecting clauses to move from P to O using rule 9.

Formally, a derivation is a possibly infinite sequence of states S₀ ⊢ S₁ ⊢ S₂ ⊢ ⋯. In Isabelle, this is expressed by the codatatype of lazy lists:
codatatype 'a llist = LNil | LCons 'a ('a llist)
Lazy list operation names are prefixed by an L or l to distinguish them from the corresponding operations on finite lists. For example, lhd xs yields xs’s head (if xs ≠ LNil), and lnth xs i yields the (i + 1)st element of xs (if i < |xs|).

We capture the mathematical notation \( S_0 \leadsto S_1 \leadsto \cdots \) formally as chain \( (\leadsto_w) S \), where \( S \) is a lazy list of states and chain is a coinductive predicate:

\[
\text{coinductive \ chain :: } (\text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}) \Rightarrow (\text{'}a \text{ list \Rightarrow bool where})
\]
\[
\text{chain } R \quad \text{(LCons \times LNil)}
\]
\[
\mid \text{chain } R \text{ xs } \land \text{ R x } \quad (\text{lhd } \text{xs}) \quad \Rightarrow \quad \text{chain } R \quad \text{(LCons \times LNil)}
\]


Coinduction is used to allow infinite chains. The base case is needed to allow finite chains. Chains cannot be empty.

Another important notion is that of the limit of a sequence \( Xs \) of sets. It is defined as the set of elements that are members of all positions of \( Xs \) except for at most finite prefix:

\[
\text{definition \ Liminf :: } \text{'}a \text{ set \ list } \Rightarrow \text{'}a \text{ set where}
\]
\[
\text{Liminf } Xs = \bigcup_{i < |Xs|} \bigcap_{j \geq i < |Xs|} \text{lnth } Xs \ j
\]

Liminf and other operators working on clause sets are lifted pointwise to states. For example, the limit of a sequence of states is defined as \( \text{Liminf } S \) = \( (\text{Liminf } Ns, \text{Liminf } Ps, \text{Liminf } Os) \), where \( Ns, Ps, \) and \( Os \) are the projections of the \( N, P, \) and \( O \) components of \( S \).

The soundness theorem states that if \( \text{RP} \) derives \( \bot \) (i.e., 0) from a set of clauses, that set must be unsatisfiable:

\[
\text{theorem \ RP_sound:}
\]
\[
\theta \in \text{clss_of } (\text{Liminf } S) \Rightarrow
\]
\[
\neg \text{ satisfiable } \left(\text{grounding_of } (\text{lhd } Ss)\right)
\]

In the above, \( \text{clss_of } (N, P, O) = N \cup P \cup O \).

A stronger, finer-grained notion of soundness relates models before and after a transition:

\[
\text{theorem \ RP_model:}
\]
\[
S \leadsto S' \Rightarrow
\]
\[
(l \models \text{grounding_of } S' \iff t \models \text{grounding_of } S)
\]

The canonical way of expressing the unsatisfiability of a set or multiset of first-order clauses with respect to Herbrand interpretations is as the unsatisfiability of its grounding.

Completeness of the prover can only be guaranteed when its rules are executed in a fair order, such that clauses do not get stuck forever in \( N \) or \( P \). Accordingly, fairness is defined as \( \text{Liminf } Ns = \text{Liminf } Ps = \emptyset \). The completeness theorem states that the limit of a fair derivation \( S \) is saturated:

\[
\text{theorem \ RP_saturated_if_fair:}
\]
\[
\text{fair } S \Rightarrow \text{ saturated_upto } (\text{Liminf } \text{grounding_of } S)
\]

In particular, if the initial problem is unsatisfiable, \( \bot \) must appear in the \( O \) component of the limit of any fair derivation:

\[
\text{corollary \ RP_complete_if_fair:}
\]
\[
\text{fair } S \leadsto \neg \text{ satisfiable } \left(\text{grounding_of } (\text{lhd } Ss)\right) \Rightarrow
\]
\[
\theta \in O \text{ of } (\text{Liminf } S)
\]

4 Ensuring Fairness

The second refinement layer is the prover \( \text{RP}_w \), which ensures fairness by assigning a weight to every clause and by organizing the set of processed clauses—the \( P \) state component—as a priority queue, where lighter clauses are chosen first. By assigning somewhat heavier weights to newer clauses, we can guarantee that all derivations are fair.

Another necessary ingredient for fairness is that derivations must be complete. For example, the incomplete derivation consisting of the single state \( (\{C\}, \emptyset, \emptyset) \) is not fair. This requirement is expressed formally as \( \text{full_chain } (\leadsto_w) S \).

For the rest of this section, we fix a full chain \( S \) such that \( P \text{of } (\text{lhd } S) = O \text{of } (\text{lhd } S) = \emptyset \).

Because each \( \text{RP}_w \) rule corresponds to an \( \text{RP} \) rule, it is straightforward to lift the soundness and completeness results from \( \text{RP} \) to \( \text{RP}_w \). The main difficulty is to show that the priority queue ensures fairness of full derivations, which is needed to obtain an unconditional completeness theorem for \( \text{RP}_w \) without the assumption fair \( S \).

Definition. The weight of a clause \( C \), which defines its priority in the queue, may depend both on the clause itself and on when it was generated. To reflect this, the \( \text{RP}_w \) prover represents clauses by a pair \( (C, i) \), where \( i \) is the timestamp.

The larger the timestamp, the newer the clause. A state is now a quadruple \( S = (N, P, O, t) \), where the first three components are finite multisets and \( t \) is the timestamp to assign to the next generation of clauses. Formally, we have the following type abbreviations:

\[
\text{type_synonym } \text{\'a wclause } = \text{\'a clause } \times \text{nat}
\]

\[
\text{type_synonym } \text{\'a wstate } =
\]
\[
\text{\'a wclause multiset } \times \text{\'a wclause multiset } \times \text{\'a wclause multiset } \times \text{nat}
\]

We extend the \text{FO_resolution_prover} locale, in which \( \text{RP} \) is defined, with a weight function that, for any given clause, is strictly monotone with respect to the timestamp, so that older copies of a clause are preferred to newer ones:

\[
\text{locale weighted_FO_resolution_prover } =
\]
\[
\text{FO_resolution_prover } +
\]

\[
\text{fixes \ weight :: } \text{\'a wclause } \Rightarrow \text{nat}
\]

\[
\text{assumes } i < j \Rightarrow \text{weight } (C, i) < \text{weight } (C, j)
\]

The weight function is otherwise arbitrary. This gives nearly unlimited freedom when selecting clauses, which is possibly the most crucial heuristic in modern provers [43]. For example, breadth-first search corresponds to the instance where \( \text{weight } (C, i) \) is defined as \( i \).

The \( \text{RP}_w \) prover uses \( \text{\'a wclause} \) for clauses. It is defined inductively as follows:

\[
\text{inductive } \leadsto_w : \text{\'a wstate } \Rightarrow \text{\'a wstate } \Rightarrow \text{bool where}
\]
\[
\text{Neg } A \in C \land \text{Pos } A \in C \Rightarrow
\]
\[
(N \cup \{ (C, t) \}, P, O, t) \leadsto_w (N \cup P \cup O, t)
\]
\[
| D \in \text{fst } (P \cup O) \land \text{subsumes } D C \Rightarrow
\]
\[
(N + \{ (C, i) \}, P, O, t) \leadsto_w (N, P, O, t)
\]
A Verified Prover Based on Ordered Resolution

CPP ‘19, January 14–15, 2019, Cascais, Portugal

where

\[ D \in \text{fist} \{N \wedge C \in \text{fist} \mathcal{P} \wedge \text{strictly-subsumes} \} \]  
\[ D \in \text{fist} \{N \wedge \text{strictly-subsumes} \} \]  
\[ (N', \mathcal{P}, \mathcal{O}, t) \rightarrow_{w} (N, \{E, k \in \mathcal{P} \wedge E \neq C\}, \mathcal{O}, t) \]  
\[ (N, \mathcal{P}, \mathcal{O} \cup \{(C, i)\}, t) \rightarrow_{w_{t}} (N', \mathcal{P}, \mathcal{O}, t) \]  
\[ (N, \mathcal{P} \cup \{(C, i)\}, \mathcal{O}, t) \rightarrow_{w_{t}} (N', \mathcal{P} \cup \{(C, i)\}, \mathcal{O}, t) \]  
\[ (N \cup \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightarrow_{w_{t}} (N', \mathcal{P} \cup \{(C, i)\}, \mathcal{O}, t) \]  
\[ (\forall (j, d) \in \mathcal{P}, \text{weight} (C, i) \leq \text{weight} (D, j)) \wedge \]  
\[ \mathcal{N} = \text{mset_set} ((\lambda D. (D, t)) \cdot \text{concl_of} \cdot \text{infers_between} \]  
\[ \text{set_mset (fist } 0) \} \]  
\[ (0, \mathcal{P} \cup \{(C, i)\}, t) \rightarrow_{w_{t}} \]  
\[ (N', \{(D, j) \in \mathcal{P} : D \neq C\}, \mathcal{O} \cup \{(C, i)\}, t + 1) \]

where fist is the function that returns the first component of a pair, mset_set converts a set to the multiset with exactly one copy of each element in the set, and set_mset converts a multiset to the set of elements in the multiset.

The most important differences with RP are in the last transition rule. This rule, which computes inferences, assigns timestamp t to each newly computed clause D and increments t. Moreover, since we want \( \mathcal{P} \) to work as a priority queue, \( \mathcal{P}_{\text{w}} \) chooses a clause \( C \) with the smallest weight.

Another difference is that \( \mathcal{P}_{\text{w}} \) uses finite multisets for representing \( N, \mathcal{P}, \) and \( \mathcal{O} \). They offer a compromise between sets in layer 1 and lists in layer 3. Finite multisets also help eliminate some unfair derivations. Finiteness guarantees that each clause in \( N \) gets the opportunity to move to \( \mathcal{P} \) (and further to \( \mathcal{O} \). Moreover, whereas the based-set RP allows idle transitions, such as \( (N \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightarrow (N, \mathcal{P} \cup \{C\}, \mathcal{O}) \) \( \mathcal{C} \in N \cap \mathcal{P} \), the use of multisets and \( \cup \) precludes such transitions in \( \mathcal{P}_{\text{w}} \).

Timestamps are preserved when clauses are moved between \( N, \mathcal{P}, \) and \( \mathcal{O} \). They are also preserved by reduction steps (rules 5 to 7). This works because reduction can only take place finitely many times—a k-literal clause can be reduced at most k times. Therefore, there is no risk of divergence due to an infinite chain of reductions.

Timestamps introduce a new danger. It may be the case that a clause \( C \) is in the limit if we project away the timestamps, but that no single timestamped clause \( (C, i) \) belongs to the limit because the timestamps keep changing, as in the infinite sequence \( \{(C, 0)\}, \{(C, 1)\}, \{(C, 2)\}, \ldots \). This could in principle arise due to subsumption, leading to derivations such as the following:

\[ \land \land \{(C, 0)\}, \ldots \rightarrow_{w} \land \land \{(C, 0), (C, 1)\}, \ldots \rightarrow_{w} \]  
\[ \land \land \{(C, 1)\}, \ldots \rightarrow_{w} \land \land \{(C, 1), (C, 2)\}, \ldots \rightarrow_{w} \]  
\[ \land \land \{(C, 2)\}, \ldots \rightarrow_{w} \ldots \]

To prevent this, the \( \mathcal{P}_{\text{w}} \) rules are formulated so that whenever they remove the earliest copy of any clause \( C \in \mathcal{P} \), they also remove all its copies from \( \mathcal{P} \). This property is captured by the following lemma:

**Lemma preserve_min_P:**

\[ S \rightarrow_{w} S' \wedge (C, i) \in \mathcal{P} \text{ of } S \wedge C \in \text{fist } \mathcal{P} \text{ of } S' \]  
\[ \wedge (\forall k. (C, k) \in \mathcal{P} \text{ of } S \implies k \geq i) \implies \]  
\[ (C, i) \in \mathcal{P} \text{ of } S' \]

This completes our review of \( \mathcal{P}_{\text{w}} \). As an intermediate step towards a more concrete prover, we restrict the weight function to be a linear equation that considers both timestamps and clause sizes:

**locale weighted_FO_resolution_prover_with_size:**

\[ \mathcal{P}_{\text{w}} \text{ resolution_prover +} \]

**fixes** size_factor :: nat and timestamp_factor :: nat

**assumes** timestamp_factor > 0

**begin**

**fun** weight :: 'a wclause => nat where

weight \( (C, i) \) = size_factor \* |C| + timestamp_factor \* i

**end**

where \( |C| = \sum_{A : \text{AE} \in C} \sim_{\text{AE}} |A| \). It is easy to prove that this definition of weight is strictly monotone and hence that this locale is a sublocale of \( \mathcal{P}_{\text{w}} \text{ resolution_prover} \). This gives us a correspondingly specialized version of \( \mathcal{P}_{\text{w}} \), which will form the basis of further refinement steps.

The idea of organizing \( \mathcal{P} \) as a priority queue is well known in the automated reasoning community. Bachmair and Ganzinger [2, p. 44] mention it in a footnote, but they require the weight to be monotone not only in the timestamp but also in the clause size, claiming that this is necessary to ensure fairness. Our proof reveals that clause size is irrelevant, even in the presence of reductions. This demonstrates how working out the details and making all assumptions explicit using a proof assistant can help clarify fine theoretical points.

**Refinement Proofs.** To lift the soundness and completeness results about RP to \( \mathcal{P}_{\text{w}} \), we must first show that any possible behavior of \( \mathcal{P}_{\text{w}} \) on states of type \( \text{wstate} \) is a possible behavior of RP on the corresponding values of type \text{state}:

**Lemma weighted_RP_imp_RP:**

\[ S \rightarrow_{w} S' \Rightarrow \text{state of } S \rightarrow \text{state of } S' \]

The proof is by induction on the rules of \( \mathcal{P}_{\text{w}} \), with one difficult case. Inference computation (rule 9) converts a set to a finite multiset using mset_set, which is undefined for infinite sets. Thus, we must show only a finite set of inferences may be performed from a finite clause set:

**Lemma finite_ord_FO_resolution_inferences_between:**

\[ \text{finite } D \Rightarrow \text{finite (infers_between } D C) \]

A binary resolution inference takes two premises, of the form \( \text{CAA} = C \lor A_{1} \lor \cdots \lor A_{k} \) and \( \text{DA} = \neg A \lor D \), and produces a conclusion \( E = (C \lor D) \cdot \sigma \). It can be represented compactly by a tuple of the form \( \langle \text{CAA}, \text{DA}, \text{AA}, \text{A}, \text{E} \rangle \), where \( \text{AA} = A_{1} \lor \cdots \lor A_{k} \). We must show that the set of such
tuples produced by 

\[ \text{infers\_between} \] is finite, assuming \( D \) is finite. First, observe that the last component \( E \) of a tuple is determined by the other four. Hence it suffices to consider quadruples \( (CA, DA, AA, A) \). Let \( DC = D \cup \{C\} \), and let \( n \) be the length of the longest clause in \( DC \). Moreover, let \( A = \bigcup_{D \in DC} \text{atoms\_of} \ D \) and \( AA = \{B | \text{set\_mset} \ B \subseteq A \land |B| \leq n \} \). Then all inferences between \( D \) and \( C \) belong to \( DC \times DC \times AA \times AA \), a cartesian product of finite sets.

**Soundness and Completeness Proofs.** Using the refinement lemma \( \text{weighted\_RP\_imp\_RP} \), it is easy to lift the \( \text{RP\_model} \) theorem (Section 3) to \( \text{RP}_w \):

**theorem** \( \text{weighted\_RP\_model} \):

\[ S \prec \text{grounding\_of} \ S' \iff I \models \text{grounding\_of} \ S \]

Completeness is considerably more difficult. We first show that the use of timestamps ensures that all full \( \text{RP}_w \) derivations are fair. In principle, a full derivation could be unfair by virtue of being finite and ending in a state such as \( N' \) or \( P \) is nonempty. However, this is impossible because a transition of rule 8 or 9 could then be taken from the last state, contradicting the hypothesis that the derivation is full. Hence, finite full derivations are necessarily fair:

**lemma** \( \text{fair\_if\_finite} \):

\[ \text{finite} \ S_1 \iff \text{fair} \ (\text{lmap} \ \text{state\_of} \ S_1) \]

There are two ways in which an infinite derivation \( S_1 \) in \( \text{RP}_w \) could be unfair: A clause could get stuck forever in \( N' \), or in \( P \). We show that the \( N' \) case is impossible by defining a measure on states that decreases with respect to the lexicographic extension of \( > \) on \text{nat} to pairs:

**abbreviation** \( \text{RP\_basic\_measure} :: 'a \text{\_wstate} \Rightarrow \text{nat}^2 \)

where

\[ \text{RP\_basic\_measure} \ (N', P, O, t) \equiv \text{sum} \ ((\lambda(C, _). |C| + 1 | (N' \cup P \cup O)), |N'|) \]

The first component of the pair is the total size of all the clauses in the state, plus 1 for each clause to ensure that empty clauses are also counted. The second component is the number of clauses in \( N' \). It is easy to see why the measure is decreasing. Rule 9, inference computation, is not applicable due to our assumption that a clause remains stuck in \( N' \). Rule 8, which moves a clause from \( N' \) to \( P \), decreases the measure’s second component while leaving the first component unchanged. The other rules decrease the first component since they remove clauses or literals. Formally:

**lemma** \( \text{weighted\_RP\_basic\_measure\_decreasing\_N} \):

\[ S \prec \text{natLess \ \langle lex \rangle \ \text{natLess \ \langle lex \rangle}} \text{nat\_Less} = ((m, n) | m < n) \]

What if a clause \( C \) is stuck in \( P \)? Lemma \( \text{preserve\_min\_P} \) states that in any step, either all copies of \( C \) are removed or the one with the lowest timestamp is kept. Hence, \( C' \)’s timestamp will either remain stable or decrease over time. Since \( > \) is well founded on natural numbers, eventually a fixed \( i \) will be reached and will belong to the limit:

**lemma** \( \text{persistent\_wclause\_in\_P\_if\_persistent\_clause} \):

\[ C \in \text{Liminf} \ (\text{lmap} \ P \ \text{of} \ (\text{lmap} \ \text{state\_of} \ S_i)) \Rightarrow \exists i. (C, i) \in \text{Liminf} \ (\text{lmap} \ \text{set\_mset} \ \text{of} \ P \ \text{of} \ S_i) \]

Again, we define a measure, but it must also decrease when inferences are computed and new clauses appear in \( N \). (In this case, \( \text{RP\_basic\_measure} \) may increase.) Our new measure is parameterized by a predicate \( p \) that can be used to filter out undesirable clauses:

**abbreviation** \( \text{RP\_filtered\_measure} :: \ ('a \text{\_wclause} \Rightarrow \text{bool}) \Rightarrow 'a \text{\_wstate} \Rightarrow \text{nat}^2 \text{\_\_filtered} \)

where

\[ \text{RP\_filtered\_measure} \ (N, P, O, t) \equiv (\text{sum} \ ((\lambda(C, _). |C| + 1 | (N' \cup P \cup O | p \ D_i)), |D_i| \in |N' | \ p \ D_i)), |D_i| \in |P | \ p \ D_i) \]

Notice that the case \( \text{RP\_filtered\_measure} \ (\lambda\ _ {\_}. \text{True}) \) essentially amounts to \( \text{RP\_basic\_measure} \). In the formalization, we use \( \text{RP\_filtered\_measure} \ (\lambda\ _ {\_}. \text{True}) \) to avoid duplication.

Suppose the clause \( C \) that is stuck in \( P \) has weight \( w \) in the limit, and suppose that a clause \( D \) is moved from \( P \) to \( O \) by rule 9. That clause’s weight must be at most \( w \); otherwise, it would not have been preferred to \( C \). Thus, infinite derivations necessarily consist of segments each consisting of finitely many applications of rules other than rule 9 followed by an application of rule 9: \( (\prec \ _ {\_} \prec \ _ {\_} \ _ {\_} \ _ {\_} \prec \ _ {\_})^n \). Since each application of rule 9 increases the \( t \) component of the state, eventually we reach a state in which \( t > w \). As a consequence of strict monotonicity of weight, any clauses generated by inference computation from that point on will have weights above \( C \)’s, and if \( C \) remains stuck, then so must these clauses. Thus, we can ignore these clauses altogether, by using \( \lambda(C, i). i \leq w \) as the filter \( p \). We adapt the corresponding relation to consider the extra argument:

**abbreviation** \( \text{RP\_filtered\_rel} :: (\text{nat}^2)^2 \ \text{set \ where} \)

\[ \text{RP\_filtered\_rel} \equiv \text{nat\_Less \ \langle lex \rangle \ \text{nat\_Less \ \langle lex \rangle} \ \text{nat\_Less} \]

The measure \( \text{RP\_filtered\_measure} \ (\lambda\ _ {\_}. i. i \leq w) \) decreases for steps occurring between inference computations and for all steps once we have reached a state where \( t > w \) (at which point all inference computations are blocked by \( C \)). To obtain a measure that also decreases on inference computation, we add a component \( w + 1 - t \) to the measure. We also add a component \( \text{RP\_basic\_measure} \) to ensure that the measure decreases when a clause \( (C, i) \) such that \( i > w \) is simplified. This yields the combined measure:

**abbreviation** \( \text{RP\_combined\_measure} :: \ ('a \text{\_wstate} \Rightarrow \text{nat} \times \text{nat}^2 \times \text{nat}^3 \text{\_\_combined} \)

where

\[ \text{RP\_combined\_measure} \ w \ S \equiv (w + 1 - t \ \text{of} \ S), \text{\_\_filtered} \text{\_measure} (\lambda\ _ {\_}. i. i \leq w) \ S \), \text{\_\_basic\_measure} \ S \]
This measure is indeed decreasing with respect to a left-to-right lexicographic order:

**lemma** `weighted_RP_basic_measure_decreasing_P`:

\[ S \rightsquigarrow_w S' \land C_i \in \mathcal{P}_o(S) \implies \]
\[ \text{(RP\_combined\_measure (weight } C_i) S', \]
\[ \text{RP\_combined\_measure (weight } C_i) S) \]
\[ \in \text{natLess } <\text{lex}> \text{ RP\_filtered\_rel }<\text{lex}> \text{ RP\_basic\_rel} \]

By combining the two lemmas `weighted_RP_basic_measure_decreasing_N` and `weighted_RP_basic_measure_decreasing_P`, we can prove all derivations starting with \( \mathcal{P} = \mathcal{O} = \emptyset \) fair:

**theorem** `weighted_RP_fair`:

fair (limstate of \( \mathcal{S} \))

Since all derivations are fair and \( \text{RP}_w \) derivations correspond to RP derivations, it is trivial to lift RP’s saturation and completeness theorems:

**corollary** `weighted_RP_saturated`:

\( \text{saturated\_upto (Liminf (limgrounding of } \mathcal{S} ))} \)

**corollary** `weighted_RP_complete`:

\( \Rightarrow \text{ satisfiable (grounding of (ld } \mathcal{S} ))} \)

\( \emptyset \in \text{O\_of (Liminf (limstate of } \mathcal{S} ))} \)

5 Eliminating Nondeterminism

The third refinement layer defines a functional program \( \text{RP}_d \) that embodies a specific rule application strategy, thereby eliminating \( \text{RP}_w \)’s nondeterminism. Clauses are represented by lists, and multisets of clauses by lists of lists.

**Definition.** Our prover corresponds roughly to the following pseudocode:

```haskell
function \( \text{RP}_d(N, \mathcal{P}, \mathcal{O}, t) \) is
repeat forever
  if \( \bot \in \mathcal{P} \cup \mathcal{O} \) then
    return \( \mathcal{P} \cup \mathcal{O} \)
  else if \( N = \mathcal{P} = \emptyset \) then
    return \( \mathcal{O} \)
  else if \( N = \emptyset \) then
    let \( C \) be a minimal-weight clause in \( \mathcal{P} \); \( N := \text{conclusions of all inferences from } \mathcal{O} \cup \{C\} \)
    involving \( C \), with timestamp \( t \);
    move \( C \) from \( \mathcal{P} \) to \( \mathcal{O} \);
    \( t := t + 1 \)
  else
    remove an arbitrary clause \( C \) from \( N \);
    reduce \( C \) using \( \mathcal{P} \cup \mathcal{O} \);
    if \( C = \bot \) then
      return \( \{\bot\} \)
    else if \( C \) is neither a tautology nor subsumed by a clause in \( \mathcal{P} \cup \mathcal{O} \) then
      reduce \( \mathcal{P} \) using \( C \);
      reduce \( \mathcal{O} \) using \( C \), moving any reduced clauses from \( \mathcal{O} \) to \( \mathcal{P} \);
      remove all clauses from \( \mathcal{P} \) and \( \mathcal{O} \) that are strictly subsumed by \( C \);
      add \( C \) to \( \mathcal{P} \)

The function should be invoked with \( N \) as the input problem, \( \mathcal{P} = \mathcal{O} = \emptyset \), and an arbitrary timestamp \( t \) (e.g., 0). The loop is loosely modeled after Vampire’s proof procedure [52].

In Isabelle, the list-based representations compel us to introduce the following type abbreviations:

**type_synonym** ‘a lclause = ‘a literal list

**type_synonym** ‘a dclause = ‘a lclause × nat

**type_synonym** ‘a dstate =

‘a dclause list × ‘a dclause list × ‘a dclause list × nat

The prover is defined inside a locale that inherits `weighted\_FO\_resolution\_prover\_with\_size\_timestamp\_factors`. The core function, \( \text{RP}_d\_step \), performs a single iteration of the main loop. Here is the definition, excluding auxiliary functions:

```haskell
fun \( \text{RP}_d\_step :: ’a dstate ⇒ ’a dstate where} \)
\( \text{RP}_d\_step (N, \mathcal{P}, \mathcal{O}, t) = \)
if \( \exists C_i \in \mathcal{P} @ \mathcal{O} \). \text{fst } C_i = [] \) then
  (\\[[],\[]\], \text{remdup } \mathcal{P} @ \mathcal{O}, \text{t }+|\text{remdup } \mathcal{P}|)
else case \( N \) of
  [] ⇒
  (case \( \mathcal{P} \) of
    [] ⇒ (N, \mathcal{P}, \mathcal{O}, t)
    | \( \mathcal{P}_0 \# \mathcal{P}’ \) ⇒
      let
      \( (C, i) = \text{select\_min\_weight\_clause } \mathcal{P}_0 \mathcal{P}’; \)
      \( N = \text{map } (\lambda D \cdot (D, i)) \) (\text{remdup}
      \text{resolve\_rename } C \mathcal{C} \mathcal{C} @ \text{concat } (\text{map}
      \text{resolve\_rename\_both\_ways } C \circ \text{fst } O));
      \( \mathcal{P} = \text{filter } (\lambda (D, j). \text{mset } D \neq \text{mset } C) \mathcal{P}; \)
      \( \mathcal{O} = (C, i) \# \mathcal{O}; \)
      \( t = t + 1 \)
    \)
  \)
\( N, \mathcal{P}, \mathcal{O}, t) \)
\( | (C, i) \# N \Rightarrow \)
  let \( C = \text{reduce } (\text{map } \text{fst } (\mathcal{P} @ \mathcal{O})) \) [] \) in
  if \( C = [] \) then
  (\\[[],\[]\], \text{t }+1)
  else if \text{is\_tautology } C
  \( \forall \text{subsume } (\text{map } \text{fst } (\mathcal{P} @ \mathcal{O})) C \) then
  (N, \mathcal{P}, \mathcal{O}, t)
  else let
  \( \mathcal{P} = \text{reduce\_all } C \mathcal{P}; \)
  \( \text{back\_to } \mathcal{P}, \mathcal{O} = \text{reduce\_all2 } C \mathcal{O}; \)
  \( \mathcal{P} = \text{back\_to } \mathcal{P} @ \mathcal{P}; \)
  \( \mathcal{O} = \text{filter } ((\neg ) \circ \text{strictly\_subsume } C \circ \text{fst}) \mathcal{O}; \)
  \( \mathcal{P} = \text{filter } ((\neg ) \circ \text{strictly\_subsume } C \circ \text{fst}) \mathcal{P}; \)
  \( \mathcal{P} = (C, i) \# \mathcal{P} \)
  \)
  (N, \mathcal{P}, \mathcal{O}, t)
\)

The # operator abbreviates the Cons constructor, and @ is the append operator.

The existential quantifier above is unproblematic because it ranges over a finite set, but some of the auxiliary functions use infinite quantification. Notably, subsumption of a clause
where $\alpha \cdot \sigma \subseteq D$ (Section 2), where $\sigma$ ranges over substitutions. Nonexecutable constructs are acceptable if we know that we can replace them by equivalent executable constructs further down the refinement chain; for example, an implementation of substitution can compute a witness $\sigma$ using matching, instead of blindly enumerating all possible substitutions.

The main program is a tail-recursive function that repeatedly calls $RP_d_step$ until a final state $([], [], O, t)$ is reached, at which point it returns the set $O$ stripped of its timestamps:

```
partial_function (option)
RP_d :: 'a dstate ⇒ 'a lclause list option
where
  RP_d S = if is_final S then Some (map fst (O_of S))
  else RP_d (RP_d_step S)
```

Since the recursion may diverge, we cannot introduce the function using the `fun` command [22]. Instead, we use `partial_function (option)` [23], which puts the computation in an option monad. The function’s result is of the form Some $R$ if the recursion terminates and None otherwise.

**Refinement Proofs.** Using refinement, we connect the $RP_d$-step function to the $RP_w$ predicate. $RP_d_step$ has a coarser granularity than $RP_w$: A single invocation on a nonfinal state $S$ can amount to a chain of $RP_w$ transitions. This is captured by the following (weak)-refinement property:

```
lemma nonfinal_deterministic_RP_step:
  ~ is_final S ⇒
  wstate_of S ~ w\_w5 wstate_of (RP_d_step S)
```

where $wstate_of$ converts $RP_d$ states to $RP_w$ states. The entire proof, including key lemmas, is about 1300 lines long. It follows the case distinctions in $RP_d_step$’s definition:

**case** $\exists C \in \mathcal{P} \cap O$. $\mathit{fst} C = \mathbb{I}$:

- By induction on $\mathit{remdups} \mathcal{P}$ (where remdups removes duplicates), there must exist a derivation of the form
  
  $\begin{align*}
  & wstate_of (N', P, O, t) \\
  \sim w_2 & \langle \mathit{wstate_of (P, O, t)} \rangle \sim w_9 & \langle \mathit{wstate_of (N', P', (C, i) \# O, t + 1)} \rangle \sim w & \langle \mathit{wstate_of ([], [], O', t + |\mathit{remdups P'}|)} \rangle
  \end{align*}$

  for $P'$ = filter $(\lambda (D, j), \text{mset } D \neq \text{mset } C) \mathcal{P}, O' = \mathit{remdups P' \cap O}$, and suitable $N'$ and $(C, i) \in \mathcal{P}$. The last step is justified by the induction hypothesis.

**case** $N' = \mathcal{P} = \mathbb{I}$:

- Contradiction with the assumption that $(N', P, O, t)$ is a nonfinal state.

**case** $N' = [\mathbb{I}]$:

- It suffices to show that the transition
  
  $\begin{align*}
  & wstate_of ([], P, O, t) \\
  \sim w_5 & \langle \mathit{wstate_of (N', P', (C, i) \# O, t + 1)} \rangle
  \end{align*}$

  is possible, where $(C, i) \in \mathcal{P}$ is a minimal-weight clause, $N' = \mathit{map (\lambda D. (D, t)) (remdups (\mathit{resolve_rename C C})}$

  @ concat (map (resolve_rename_both_ways C @ fst) $O)$), and $P' = \mathit{filter (\lambda (D, j), \text{mset } D \neq \text{mset } C) \mathcal{P}$

  The main proof obligation is that $N'$, converted to multisets, equals the multiset $\text{mset} ((ID. (D, t)) \sim \mathit{cond_of_infers_between (set_mset (fst \langle O \rangle)}) C)$ specified in rule $\sim \omega_5$. The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

**otherwise:**

- Let $C' = \mathit{reduce (map \mathcal{P} \mathit{map \mathcal{F} \cap O}) \mathbb{I} C$. If $C' = \mathbb{I}$, then
  
  $\begin{align*}
  & wstate_of ((C, i) \# N', \mathcal{P}, O, t) \sim \omega_5 wstate_of (([], i) \# N, \mathcal{P}, O, t) \sim \omega_3 wstate_of (([], i) \# N', \mathcal{P}, O, t) \sim \omega_4 wstate_of (([], i) \# N', \mathcal{P}, O, t) \sim \omega_8 wstate_of (N', ([], 0), [t, t]) \sim \omega_2 wstate_of (\mathbb{I}, ([], 0), [t, t]) \sim \omega_9 wstate_of (\mathbb{I}, ([], 0), [t, t])
  \end{align*}$

  Otherwise, if is_tautology $C' \lor \text{subsume (map \mathcal{P} \mathit{map \mathcal{F} @ O})} C'$, then
  
  $\begin{align*}
  & wstate_of ((C, i) \# N', \mathcal{P}, O, t) \sim \omega_5 wstate_of ((C', i) \# N', \mathcal{P}, O, t) \sim \omega_3 wstate_of (N', (C', i) \# N', \mathcal{P}, O, t)
  \end{align*}$

  Otherwise:

  $\begin{align*}
  & wstate_of ((C, i) \# N', \mathcal{P}, O, t) \sim \omega_5 wstate_of ((C', i) \# N', \mathcal{P}, O, t) \sim \omega_3 wstate_of ((C', i) \# N', \mathcal{P}', O', t) \sim \omega_4 wstate_of ((C', i) \# N', \mathcal{P'}, O', t) \sim \omega_9 wstate_of ((C', i) \# N', \mathcal{P''}, O'', t)
  \end{align*}$

  for suitable lists $P'$, $\text{back_to} \mathcal{P}, O', O''$, and $P''$.

**Soundness and Completeness Proofs.** Let $S_0 = (N_0, [], [], 0)$ be an arbitrary initial state. Soundness means that whenever $RP_d S_0$ terminates with some clause set $R$, then $R$ is a saturation that satisfies the same models as $N_0$. Moreover, if $N_0$ is unsatisfiable, then $R$ contains $\perp$, providing a simple syntactic check for unsatisfiability. Completeness means that divergence is possible only if $N_0$ is satisfiable. For satisfiable clause sets $N_0$, both termination and divergence are possible.

To lift soundness and completeness results from $RP_w$ to $RP_d$, we first define $SS$ as a full chain of nontrivial $RP_d$ steps starting from $S_0$. We let $SS = derivation_from S_0$, with

```
princorec derivation_from :: 'a dstate ⇒ 'a dstate list
where
  derivation_from S = LCons S (if is_final S then LNil
else derivation_from (RP_d_step S))
```

Based on $SS$, we let $\omega SS = \mathit{lmap wstate_of SS}$ and note that $\omega SS$ is a full chain of “big” $\sim \omega_5$ steps. Using a lemma that will be proved below, we obtain a full chain $\mathit{ss wSS}$ of “small”
The definition of $\text{RP}_d$, using \textbf{partial function}, gives us an induction rule restricted to the case where $\text{RP}_d$ terminates (i.e., returns a Some value). This rule can be used to prove that $\mathcal{S}$ and hence $w\mathcal{S}$ and $ssw\mathcal{S}$ are finite sequences.

Soundness takes the form of a pair of theorems that lift weighted $\text{RP}_d$ model and weighted $\text{RP}_s$:

\begin{itemize}
  \item \textbf{Theorem deterministic $\text{RP}_d$ model:}
  \end{itemize}

\begin{itemize}
  \item \begin{align*}
  I \models \text{grounding of } \mathcal{N}_0 &\iff I \models \text{grounding of } R \\
  \text{saturated upto (grounding of } R) &
  \end{align*}
  \end{itemize}

In most applications, all that matters is the satisfiability status of the set $\mathcal{N}_0$. It can be retrieved syntactically:

\begin{itemize}
  \item \textbf{Corollary deterministic $\text{RP}_d$ refutation:}
  \end{itemize}

\begin{itemize}
  \item $\neg$ satisfiable (grounding of $\mathcal{N}_0$) $\iff \emptyset \in R$
  \end{itemize}

Completeness is proved in a separate context that assumes nontermination: $\text{RP}_d \mathcal{S}_0 = \text{None}$. The strongest result we prove is that this assumption implies the satisfiability of $\mathcal{N}_0$:

\begin{itemize}
  \item \textbf{Theorem deterministic $\text{RP}_s$ complete:}
  \end{itemize}

\begin{itemize}
  \item satisfiable (grounding of $\mathcal{N}_0$)
  \end{itemize}

The proof is by contradiction:

Assume that $\neg$ satisfiable (grounding of $\mathcal{N}_0$). Hence, by \text{weighted $\text{RP}_s$ complete} we have $\emptyset \in O_{\text{of }} ssw\mathcal{S}$. It is easy to show that $ssw\mathcal{S}$’s limit is a subset of $w\mathcal{S}$’s limit; hence $\emptyset \in O_{\text{of }} w\mathcal{S}$. This implies the existence of a natural number $k$ such that $\emptyset \in O_{\text{of }} (\text{Inth } w\mathcal{S} k)$. Hence $\emptyset \in O_{\text{of }} (\text{RP}_d\text{-step } k \mathcal{S}_0)$. However, by an induction on $k$, we can show that $\text{RP}_d$ must terminate after at most $k$ iterations, contradicting the assumption that $\text{RP}_d$ diverges.

A Coinductive Puzzle. A single “big” step of the deterministic prover $\text{RP}_d$ may correspond to many “small” steps of the weighted prover $\text{RP}_w$. To transfer the results from $\text{RP}_w$ to $\text{RP}_d$, we must expand the big steps. The core of the expansion is an abstract property of chains and transitive closure:

Let $R$ be a relation and $xs$ a chain of $R^+$ transitions. There exists a chain of $R$ transitions that embeds $xs$—i.e., that contains all elements of $xs$ in the same order and with only finitely many elements inserted between each pair of consecutive elements of $xs$.

On finite chains, this property can be proved by straightforward induction. But the completeness proof must also consider infinite chains. Coinduction and corecursion up-to techniques are useful for such tasks.

The desired property is stated formally as follows:

\begin{itemize}
  \item \textbf{Lemma chain tranclp imp exists chain:}
  \end{itemize}

\begin{itemize}
  \item \begin{align*}
  \text{chain } R' &\text{ xs } \implies \\
  \exists y. \text{ chain } R \text{ ys } &\land \text{ xs } \subseteq \text{ ys } \land \text{ lhd } \text{ xs } = \text{ lhd } \text{ ys } \\
  &\land \text{ llast } \text{ xs } = \text{ llast } \text{ ys}
  \end{align*}
  \end{itemize}

where the embedding $\subseteq$ of lazy lists is defined coinductively using $\uri{+}$, which prepends a finite list to a lazy list:

\begin{itemize}
  \item \textbf{Coinductive $\subseteq ::$ a list } $\implies$ $\text{ a list } \implies$ $\text{ bool }$ where
  \end{itemize}

\begin{itemize}
  \item \begin{align*}
  \text{finite } &\text{ xs } \implies \text{ LNil } \subseteq \text{ xs } \\
  | \text{ xs } \subseteq &\text{ ys } \implies \text{ LCons } \text{ x } \text{ xs } \subseteq \text{ zs } + \text{ LCons } \text{ x } \text{ ys}
  \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{Fun $\uri{+}$ :: $\text{ a list } \implies$ $\text{ a list } \implies$ $\text{ a list }$ where}
  \end{itemize}

\begin{itemize}
  \item \begin{align*}
  [] &\uri{+} \text{ xs } = \text{ xs } \\
  | (z \# \text{ zs}) &\uri{+} \text{ xs } = \text{ LCons } z \text{ (zs } \uri{+} \text{ xs})
  \end{align*}
  \end{itemize}

The definition of $\subseteq$ ensures that infinite lazy lists only embed other infinite lazy lists, but not the finite ones: $\text{ xs } \subseteq \text{ ys } \implies \text{finite } \text{ xs } \implies \text{finite } \text{ ys}$. The unguarded calls to llast may seem worrying, but the function is conveniently defined to always return the same unspecified element for infinite lists.

To prove the lemma above, we instantiate the existential quantifier by the following corecurcively defined witness:

\begin{itemize}
  \item \textbf{Corec wit \uri{::} $\text{ (a }\uri{\Rightarrow} \text{ a }\uri{\Rightarrow} \text{ bool }$) $\implies$ $\text{ a list } \implies$ $\text{ a list }$ where}
  \end{itemize}

\begin{itemize}
  \item \begin{align*}
  \text{wit } &\text{ R } \text{ xs } = \text{ (case } \text{ xs } \text{ of} \\
  \text{ LCons } \text{ x } \text{ (LCons } &\text{ y } \text{ ys }) \implies \\
  \text{ LCons } \text{ x } \text{ (pick } \text{ R } &\text{ x } \text{ y } \uri{+} \text{ wit } \text{ R } \text{ (LCons } \text{ y } \text{ ys}) \\
  \text{ } | _{\p} &\uri{=} \text{ xs})
  \end{align*}
  \end{itemize}

Here pick $\text{ R } \text{ x } \text{ y}$ returns an arbitrary finite list of $R$-related states connecting the $R^*$-related $x$ and $y$. Its definition is pick $\text{ R } \text{ x } \text{ y } = (\text{ SOME } \text{ zs. chain } \text{ R } \text{ (llist } \text{ of } \text{ (x }\#\text{ zs } @ \text{ [y]))})$, where $\text{llist } \text{ of}$ converts finite lists into lazy lists and $\text{SOME}$ is Hilbert’s choice operator. Thus, pick satisfies the characteristic property $\text{ R }^* \text{ x } \text{ y } \implies \text{ chain } \text{ R } \text{ (llist } \text{ of } \text{ (x }\#\text{ pick } \text{ R } \text{ x } \text{ y } @ \text{ [y]))}$. The nonexecutability entailed by the use of Hilbert choice is unproblematic because the wit function is used only in the proofs and not in the prover’s code.

The definition of wit is not primitively corecursive. Although there is a guarding LCons constructor, the corecursive call occurs under $\uri{+}$, which makes the productivity of this function nontrivial. This syntactic structure of the definition is called \text{ corecursive up to } $\uri{+}$. Ultimately, wit is productive because $\uri{+}$ does not remove any LCons constructors from its second arguments. A slightly weaker requirement, called \text{ friendliness}, is supported by Isabelle’s \textbf{corec} command [5]. For the above definition to be accepted $\uri{+}$ must be registered as a “friend.” This involves a one-line proof.

The four conjuncts in $\text{chain tranclp imp exists chain}$ are discharged in turn under the assumption chain $\text{ R }^{*}$ $\text{ xs}$. In order of increasing difficulty: lhd (wit $\text{ R } \text{ xs}) = \text{ lhd } \text{ xs}$ follows by simple rewriting. Next, llast (wit $\text{ R } \text{ xs}) = \text{ llast } \text{ xs}$ requires an induction in the case of finite chains $\text{ xs}$. For any infinite chain $\text{ zs}$, llast $\text{ zs}$ is defined as a fixed unspecified $\text{ a }$ value. The properties $\text{ xs } \subseteq \text{ wit } \text{ R } \text{ xs}$ and chain $\text{ R } \text{ (wit } \text{ R } \text{ xs})$ require a coinduction on $\subseteq$ and chain, respectively. In keeping with the
where

To simplify notation, in this paper we fix the variables contained in a list of clauses. The creation of an arbitrary (but executable) linear order requires us to sort the terms belonging to set with respect to easy to translate a finite set of finite sets of terms into a finite type.

match:

 Instantiate the type unified and the well-foundedness of strict term generalization, includes a wealth of theorems, including the correctness of constraints that is compatible with a given substitution. IsaFoR includes a wealth of theorems, including the correctness of unify and the well-foundedness of strict term generalization, defined as (3σ. s · σ = t) ∧ (3σ.t · σ = s).

This infrastructure allows us to conveniently instantiate our locales substitution_ops, substitution, and mgu. We instantiate the type ‘a of atoms with term and the type ‘s of substitutions with ‘u ⇒ term and the constants ·, id, ⊖, and atm_of atms with ·, Var, α · x, x · α, and (arbitrarily) Fun 0. For the MGU computation, there is a slight type mismatch: IsaFoR offers a list-based unifier, whereas our locale requires the type term set set ⇒ (‘u ⇒ term) option. It is easy to translate a finite set of finite sets of terms into a finite list of pairs of constraints. To be executable, the translation requires us to sort the terms belonging to set with respect to an arbitrary (but executable) linear order.

Only the function renamingsapart was not present in IsaFoR. We supply a definition:

fun renamingsapart :: term clause list ⇒ (‘u ⇒ term) list
where

renamingsapart [] = []
| renamingsapart (C # Cs) =
  let as = renamingsapart Cs in
  (λv. v + max (0) vars clause list (Cs · as)) + 1) # as

where

and

Finally, the FO_resolution_prover locale requires that the type of atoms supports two comparison operators: a well-order > and a comparison ≥ that is stable under substitution (i.e., B > A ⇒ B · σ > A · σ). Moreover, > and ≥ must coincide on ground atoms. We instantiate ≥ with the Knuth–Bendix order [19] on terms, provided by IsaFoR [46]. This order is executable, stable under substitution, well-founded, and total on ground terms. The well-order >, which must be total on all terms, is then defined as an arbitrary extension of a partial well-founded order ≥ to a well-order, using Hilbert choice. This makes > nonexecutable, but this is acceptable since it is ≥, not >, that is used in the prover’s code.

Clause Subsumption. The second hurdle concerns clause subsumption. Its mathematical definition, subsumes C D ⇔ ∃σ. C · σ ⊆ D, involves an infinite quantification.

The problem of deciding whether such a substitution exists is NP-complete [18]. We start with the following naive code. In contrast to the mathematical definition, which operates on multisets of literals, our function operates on lists:

fun subsumes_list :: term literal list ⇒ term literal list ⇒ (‘u ⇒ term option) ⇒ bool
where

subsumes_list [] Ks σ = True
| subsumes_list (L # Ls) Ks σ =
  (3Ks ∈ set Ks. is_pos K = is_pos L ∧
  case match_term_list [(atm_of L, atm_of K)] σ of
  None ⇒ False
  | Some ρ ⇒ subsumes_list Ls (remove1 K Ks) ρ)

In the Cons case, we must consider all possible matching literals for L from Ks compatible with the substitution σ. The bounded existential quantification that expresses this non-determinism can be executed by iterating over the finite list Ks. The functions is_pos and atm_of are the discriminator and selector for literals. The function match_term_list is provided by IsaFoR. It attempts to extend a given substitution into Some matcher for a list of matching constraints, given as term pairs. If the extension is impossible, match_term_list returns None. This substitution-passing style is typical of purely functional implementations of matching.

It is easy to prove that the above executable function implements clauses subsumption: subsumes (mset Ls) (mset Ks) = subsumes_list Ls Ks (λx. None), where mset converts lists to multisets by forgetting the order of the elements. After the registration of this equation, Isabelle’s code generator will rewrite any code that contains the nonexecutable left-hand side to use the executable right-hand side instead.

Clause subsumption is a hot spot in a resolution prover [41]. Following Tammet [49], we implement a heuristic that often reduces the number of calls to match_term_list, which is linear in the size of the input terms, by first performing a simpler, imprecise comparison. For example, terms with different root symbols will never match, and these can be compared in
constant time. Similarly, literals with opposite polarities cannot match. We sort our (list-represented) clauses with respect to a literal quasi-order (i.e., a transitive and reflexive relation) leq_lit such that leq_lit L K only if is_pos L = is_pos K and match_term_list [(atm_of L, atm_of K)] σ = Some ρ for some σ and ρ. Any quasi-order satisfying this property can be used in a refinement of subsumes_list to remove too small literals (with respect to leq_lit), as highlighted in gray below:

\[
\text{fun subsumes_list'} :: \text{term literal list} \Rightarrow \\
\text{term literal list} \Rightarrow (\forall \Rightarrow \text{term option}) \Rightarrow \text{bool}
\]

where

\[
\begin{align*}
\text{subsumes_list'} [] \text{ Ks} & \sigma = \text{True} \\
| \text{subsumes_list'} (L \# \text{ Ls}) \text{ Ks} \sigma = \\
\text{let Ks} = \text{filter (leq_lit L) Ks in} \\
(\exists K \in \text{set Ks}. \text{ is_pos K} = \text{is_pos L} \land \\
\text{case \text{match_term_list} [(atm_of L, atm_of K)] \sigma of} \\
\text{None} & \Rightarrow \text{False} \\
| \text{Some ρ} \Rightarrow \text{subsumes_list'} \text{ Ls (remove1 K Ks) ρ}
\end{align*}
\]

The lemma subsumes_list LS KS ρ = subsumes_list’ (sort leq_lit Ls) KS ρ allows the code generator to refine the original version. In RPx, we let leq_lit be a quasi-order that (1) considers negative literals smaller than positive ones; (2) considers variables smaller than nonvariables; and (3) sorts atoms according to a total order on their root symbols.

This refinement is a local optimization: It requires us to explicitly sort one of the input clauses. A potentially more efficient refinement would be to ensure that all clauses in the prover’s state are sorted with respect to leq_lit. Sorting Ls at each invocation of subsumption could then be avoided, and filtering KS could be performed more efficiently. However, maintaining the invariant would require changes throughout the prover’s code.

**The End Result.** Finally, Isabelle can export our prover to Standard ML, Haskell, OCaml, or Scala. The command

```sml```

```
export_code RPx in SML module_name RP
```

```sml```

generates an ML module containing the implementation of our prover in about 1000 lines of code, including dependencies. The generated module exports the function RPx : (term literal list * nat) list -> bool. The input is the N component of an initial state, which consists of pairs of clauses and arbitrary timestamps (e.g., 0).

Even though in Isabelle we have proved that for any unsatisfiable input RPx will terminate and return False, the code generator guarantees only partial correctness of its output: If the ML program terminates on the ML input generated from the Isabelle term t and evaluates to the Boolean result b, the proposition RPx t = b is provable in Isabelle; by soundness, b indicates the satisfiability of the input clause set. There is recent work towards providing stronger guarantees and reducing the generator’s trusted code base [14].

**Empirical Evaluation.** To measure the gap with the state of the art, we compare our prover’s performance with that of three other provers on a benchmark suite. TPTP (Thousands of Problems for Theorem Provers) [48] is the de facto standard library for benchmarking automatic provers. We extended RPx with the trusted TPTP parser from Metis [15]. We benchmarked E 2.1, Vampire 4.2.2, Metis 2.4, and RPx on 1000 randomly selected equality-free problems from the TPTP’s FOF (first-order formulas) and CNF (first-order formulas in conjunctive normal form) categories. We converted all FOF problems to CNF using E’s classifier. Each prover was run on each problem for 60 s on an Intel Core i9-7900X (3.3 GHz 10-Core) with 128 GB of RAM.

The results are summarized in the following table, showing for each prover how many unsatisfiable and satisfiable problems were solved and how many seconds were needed on average by each prover on the problems that were solved by all four:

<table>
<thead>
<tr>
<th></th>
<th>Vampire</th>
<th>E</th>
<th>Metis</th>
<th>RPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsatisfiable</td>
<td>675</td>
<td>635</td>
<td>436</td>
<td>331</td>
</tr>
<tr>
<td>Satisfiable</td>
<td>158</td>
<td>135</td>
<td>91</td>
<td>22</td>
</tr>
<tr>
<td>Average time (s)</td>
<td>0.032</td>
<td>0.014</td>
<td>0.637</td>
<td>3.126</td>
</tr>
</tbody>
</table>

The detailed results of the evaluation are available online, together with instructions for reproducing them.3

As expected, RPx is not competitive. A prover’s performance comes from its calculus, its heuristics, and its indexing data structures. RPx employs an excellent calculus but mediocre heuristics and data structures. Better performance could be achieved by working on these last two aspects. Heuristics are often easy to verify, because their input–output specifications are permissive, but formalizing optimized data structures can be very laborious [10].

Nevertheless, sometimes the calculus is all that matters. Benchmark MSC015 from the TPTP library is a particularly challenging family Φn of first-order problems, each consisting of the following n + 2 clauses:

\[
\neg p(b, \ldots, b) \lor (p(a, \ldots, a)) \\
\neg p(a, b, \ldots, b) \lor (p(b, a, \ldots, a)) \\
\neg p(x_1, a, b, \ldots, b) \lor (p(x_1, b, a, \ldots, a)) \\
\vdots \\
\neg p(x_1, \ldots, x_{n-2}, a, b) \lor (p(x_1, \ldots, x_{n-2}, b, a)) \\
\neg p(x_1, \ldots, x_{n-2}, x_{n-1}, a) \lor (p(x_1, \ldots, x_{n-2}, x_{n-1}, b))
\]

A comment in the benchmark warns us that back in 2007, no prover could solve the Φ23 within an hour. Even in 2018, only one prover solves Φ23 within 300 s, and four provers solve Φ20 within 300 s. RPx solves Φ20 in 100 s and Φ22 in 200 s. Presumably, the reason for this success is that RPx fortuitously chooses an instance of the Knuth–Bendix order that is well suited to this benchmark.

3http://matryoshka.gforge.inria.fr/pubs/fun_rp_data.tar.gz
7 Discussion and Related Work

We found Bachmair and Ganzinger’s [2] chapter and its formalization [39, 40] suitable as a starting point for a verified prover. Nonetheless, we faced some difficulties, notably concerning the identification of suitable refinement layers. We developed layers 2, 3, and 4 largely in parallel, with each of the authors working on a separate layer. Bringing layer 2 into a state such that it both ensures fairness and could be refined further by layer 3 required several iterations.

Stepwise refinement helped us achieve separation of concerns: fairness, determinism, and executability were achieved successively. Another strength of this methodology is that it allows us to prove results at a high level of abstraction; for example, fairness is established at layer 2 already and is inherited by subsequent layers. The main difficulty with refinement is that some nontrivial machinery is necessary to lift results from one layer to the next. We believe the gain in modularity makes this worthwhile.

It took us quite some time to design a suitable measure to prove the fairness of the layer 2 prover $\text{RP}_w$. Our solution amounts to advancing to a state carrying a suitably high timestamp and filtering out all overly heavy clauses. Initially, our proof consisted of two steps—advancing and filtering—each with its own measure. This proof gave us the assurance that $\text{RP}_w$ was fair, but we found that combining the measures is both more succinct and more intelligible.

Our main objective was not to reach \textit{qed} as quickly as possible but rather to investigate how to harness a modern proof assistant to formalize the metatheory of automatic theorem provers. We found Isabelle suitable for this verification task. The Isar proof language allows us to state key intermediate steps, as in a paper proof. Standard tactics, including Isabelle’s simplifier, can be used to discharge proof obligations. The Sledgehammer tool [32] employs superposition provers and SMT (satisfiability modulo theories) solvers to swiftly identify which lemmas can be used to prove a goal; standard Isabelle tactics are then used to reconstruct the proof. Isabelle’s support for coinductive methods, including the \texttt{coinductive}, \texttt{codatatype}, and \texttt{corec} commands, helps us reason about infinite processes. Locales are a useful abstraction for defining the refinement layers. And Isabelle’s libraries, the \textit{Archive of Formal Proofs}, and IsaFoR certainly saved us months of labor.

The \textit{Archive} also includes a refinement framework [25], which has been used in a separate effort to connect the imperative code of an efficient SAT solver to an abstract calculus [6]. The framework is helpful in a variety of situations, including when the refinement relation between a concrete and an abstract data representation is not a function. But since converting a list to a multiset (between our levels 3 and 2) or a multiset to a set (between levels 2 and 1) \textit{is} a function, we did not see a need to employ it. Moreover, the framework is currently not designed for refining semidecision procedures, as acknowledged privately by its developer. We conjecture that its support for separation logic could be useful if we were to refine the prover further to obtain imperative code.

Thanks to the verification, we can trust to a very high extent that our ordered resolution prover is sound and complete. To make the prover’s performance competitive with E, SPASS, and Vampire, we would need to extend the current work along two axes. First, we should use superposition, together with its extensive simplification machinery, as the base calculus. A good starting point would be to apply our methodology to Peltier’s [33] formalization of superposition. Given that a large part of a modern superposition prover’s code consists of heuristics, which are easy to verify, the full verification of a competitive superposition prover appears to be a realistic objective for a forthcoming Ph.D. thesis. Second, the refinement chain should be continued to cover optimized algorithms and data structures. These could be specified by refining layer 4 further, along the lines of Fleury et al.’s [10] refinement of an imperative SAT solver.

In computer science, a metatheory may inspire an implementation, or vice versa, but the connection is seldom made explicit. By formalizing the metatheory, the implementation, and their connection, we can demonstrate not only the implementation’s correctness but also the metatheory’s adequacy for describing potential implementations. In particular, we have now confirmed that Bachmair and Ganzinger [2] accurately describe the abstract principles of an executable functional prover (with a few exceptions [40]), even though they provide few details beyond layer 1.

We built the prover on our earlier formalization [39, 40] of ordered resolution. Related efforts developed using Isabelle/HOL include Peltier’s [33] formalization of superposition and Schlichtkrull’s [37] formalization of unordered resolution. These developments cover only logical calculi; had we started with any of them, the first step would have been to define an abstract prover in the style of layer 1 and prove basic properties about it. Another related effort is Hirokawa et al.’s [13] formalization of ordered completion, which (like ordered resolution) can be regarded as a special case of superposition.

Formalizing a theorem proving tool using a theorem proving tool is a thrilling (if self-referential) prospect for many researchers. An early result is Ridge and Margte’s [35] verified first-order prover, based on a sequent calculus for first-order logic without full first-order terms but only variables. Kumar et al. [24] formalized the soundness of a proof assistant for higher-order logic. Jensen et al. [16] verified the soundness of a kernel for a proof assistant for first-order logic that includes a tableau prover. There are several verified SAT solvers [6, 27–29, 31, 44]. SAT being a decidable problem, termination has been proved for most solvers. First-order logic, on the other hand, is semidecidable.

A pragmatic approach to combining the efficiency of unverified code with the trustworthiness of verified code involves checking certificates produced by reasoning tools—
e.g., proofs produced by SAT solvers [9, 26]. Researchers from the first-order theorem proving community are now advocating this approach for their systems [34]. An ad hoc version of this approach is used in Sledgehammer and HOLyHammer to reconstruct proofs found by external automatic provers [4, 17].

8 Conclusion

Starting from an abstract description of an ordered resolution prover [39, 40], we verified, through a refinement chain, a purely functional prover that uses lists as its main data structure. The resulting program is interesting in its own right and could be refined further to obtain an implementation that is competitive with the state of the art.

Stepwise refinement is a keystone of our methodology, and we found it adequate. Each refinement step cleanly isolates concerns, yielding intelligible proof obligations. Refinement also helped us identify an unnecessary assumption in Bachmair and Ganzinger’s [2] chapter and clarify the argument. Lifting results from one layer to another required some thought, especially the completeness results, which correspond to liveness properties.

Having now established a methodology and built basic formal libraries, we expect that verifying other provers, using Isabelle or other systems, will be substantially easier. Because it is based on Bachmair and Ganzinger’s framework, our approach generally applies to all saturation-based provers, with or without redundancy. This includes resolution, paramodulation, ordered rewriting, superposition, and variants thereof, covering many of the most successful provers for equational [8, 12], first-order [21, 42, 53], and higher-order logic [45].

Acknowledgments

Johannes Hölzl gave us some useful advice on how to specify and reason about possibly nonterminating functions in Isabelle/HOL. Alexander Bentkamp, Mathias Fleury, Andreas Halkjær From, Carsten Fuhs, Peter Lammich, Mark Summerfield, Jörgen Villadsen, and the anonymous reviewers suggested many textual improvements. Eugene Kotelinkov, Stephan Schulz, and Geoff Sutcliffe graciously answered our questions about Vampire, E, and TPTP.

Schlichtkrull has received funding from a Ph.D. scholarship in the Algorithms, Logic and Graphs section of DTU Compute and from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 700321, LIGHTTest). Blanchette has received funding from the ERC under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). An earlier version of this paper was included as a chapter of Schlichtkrull’s Ph.D. thesis [36] with the same authors’ list.

References


