A verified prover based on ordered resolution

Schlichtkrull, Anders; Blanchette, Jasmin Christian; Traytel, Dmitriy

Published in:
Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs

Link to article, DOI:
10.1145/3293880.3294100

Publication date:
2019

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A Verified Prover Based on Ordered Resolution

Anders Schlichtkrull
Department of Computer Science
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

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 ordered resolution [2] and ordered completion [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 \( \lnot A \). An atom is a symbol applied to a tuple of terms—e.g., \( \text{prime}(n) \). 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 \( D \lor \{ \exists x. \lnot C \} \), 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 semidecidable—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,1 which aims at developing a library of results about logic and automated reasoning [3]. The Isabelle files are available in the Archive of Formal Proofs [38, 39] and in the IsaFoL repository.2 The parts specific to the functional prover refinement amount to about 4000 lines of source text. A convenient way to study the files is to open them in Isabelle/jEdit [54], as explained in the repository’s readme file. The files were created using Isabelle version 2018, but the repositories will be updated to follow Isabelle’s evolution.

2 Atoms and Substitutions

The first three refinement layers are based on an abstract library of first-order atoms and substitutions. In the fourth and final layer, the abstract framework is instantiated with concrete datatypes and functions.

We start from IsaFoL’s library of clausal logic [6], which is parameterized by a type ‘a of logical atoms. Literals L are defined as an inductive datatype: ‘a literal = Pos ‘a | Neg ‘a. The type of clauses C, D, E is introduced as the alias ‘a clause = ‘a literal multiset, where multiset is the type constructor of finite multisets. Thus, the clause ¬A ∨ B, where A and B are arbitrary atoms, is represented by the multiset {Neg A, Pos B}, and the empty clause ⊥ is represented by the empty multiset ∅. The complement operation is defined as ¬Neg A = Pos A and ¬Pos A = Neg A for any atom A.

In automated reasoning, it is customary to view clauses as multisets of literals rather than as sets. One reason is that multisets behave more naturally under substitution. For example, applying {y ⇒ x} to the two-literal clause p(x) ∨ p(y) results in p(x) ∨ p(x), which preserves the clause’s structure.

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 I |= 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 and of 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 (◦):

```isabelle
locale substitution_ops = 
  fixes id :: ‘s and ⋅ :: ‘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

```isabelle
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

```isabelle
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

```isabelle
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:

```isabelle
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) ⇒ ∃τ. is_ground τ ∧ C ⋅ τ = C ⋅ σ
```
Bachmair and Ganzinger’s nondeterministic first-order prover, called RP, captures the “dynamic” aspects of saturation. RP is defined as an inductive predicate $\leadsto$ 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 $P \cup O$ 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:

\[ \text{inductive } \leadsto :: 'a state \Rightarrow 'a state \Rightarrow bool \text{ where} \]

\begin{align*}
\text{Neg } A \in C & \land Pos A \in C \\
(N \cup \{C\}, P, O) \leadsto_1 (N, P, O) \\
D \in P \cup O & \land \text{subsumes } D \Rightarrow C \\
(N \cup \{C\}, P, O) \leadsto_2 (N, P, O) \\
D \in N & \land \text{strictly subsumes } D \Rightarrow C \\
(N, P \cup \{C\}) \leadsto_3 (N, P, O) \\
D \in N & \land \text{strictly subsumes } D \Rightarrow C \\
(N, P \cup \{C\}) \leadsto_4 (N, P, O) \\
D \in P & \land \text{reduces } D \Rightarrow C \Rightarrow L \\
(N \cup \{C \cup \{L\}\}, P, O) \leadsto_5 (N \cup \{C\}, P, O) \\
D \in N & \land \text{reduces } D \Rightarrow C \Rightarrow L \\
(N, P \cup \{C \cup \{L\}\}) \leadsto_7 (N, P \cup \{C\}) \\
(N \cup \{C\}, P, O) \leadsto_8 (N, P \cup \{C\}) \\
(0, P \cup \{C\}, O) \leadsto_9 \text{ (concl_of }' \text{ infs_between } O \cup C, P, O \cup \{C\})
\end{align*}

Subscripts on $\leadsto$ identify the rules. The notation $f \cdot X$ stands for the image of $X$ under $f$, infs_between $O \cup C$ calculates all the ordered resolution inferences whose premises are a subset of $O \cup C$ that contains $C$, and reduces $D \Rightarrow C \Rightarrow L$ is defined as $\exists D' \in \text{Pos } L' \cdot \text{R } D = D' \cup \text{Pos } L' \land L = L' \land D' \Rightarrow C$.

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

\begin{align*}
\leadsto_6 & (\theta, \emptyset, \emptyset) \\
\leadsto_9 & (\{\neg p(a, a), p(x, x), \neg p(f(x), y) \lor p(x, y)\}, \emptyset, \emptyset) \\
\leadsto_9 & (\{\neg p(a, a), p(x, x), \neg p(f(x), y) \lor p(x, y)\}, \emptyset, \emptyset) \\
\leadsto_9 & (\{\neg p(f(x), y) \lor p(x, y), p(x, x)\}) \\
\leadsto_9 & (\{\neg p(a, a), p(x, f(x))\}) \\
\leadsto_9 & (\{\neg p(f(x), y) \lor p(x, y), p(x, x)\}) \\
\leadsto_9 & (\{\neg p(f(x), y) \lor p(x, y), p(x, x)\}) \\
\leadsto_9 & \cdots
\end{align*}

We can leave $p(a, a)$ in $P$ forever and always generate more clauses of the form $p(x, f^i(x))$, 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 $S_0 \leadsto S_1 \leadsto S_2 \leadsto \cdots$. In Isabelle, this is expressed by the codatatype of lazy lists:
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 $\text{full_chain}(\sim_w) S_\emptyset$. For the rest of this section, we fix a full chain $S_\emptyset$ such that $P_{\text{of}}(\text{lh}(S)) \subseteq \text{O}_{\text{of}}(\text{lh}(S)) = \emptyset$.

Because each $\text{RP}_w$ rule corresponds to an RP rule, it is straightforward to lift the soundness and completeness results from 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 $S_\emptyset$.

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:

\[
\begin{align*}
\text{type_synonym} & \quad 'a \text{wclause} = 'a \text{clause} \times \text{nat} \\
\text{type_synonym} & \quad 'a \text{wclause multiset} = 'a \text{wclause} \times 'a \text{wclause multiset} \\
\end{align*}
\]

We extend the $\text{FO_resolution_prover}$ locale, in which 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:

\[
\begin{align*}
\text{locale} & \quad \text{weighted_FO_resolution_prover} = \\
\text{defines} & \quad \text{weight} :: 'a \text{wclause} \Rightarrow \text{nat} \\
\text{assumes} & \quad i < j \Rightarrow \text{weight}(C, i) < \text{weight}(C, j) \\
\end{align*}
\]

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 weight $(C, i)$ is defined as $i$.

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

\[
\begin{align*}
\text{inductive} & \quad \sim_w :: 'a \text{wstate} \Rightarrow 'a \text{wstate} \Rightarrow 'bool \text{ where} \\
\text{Neg} & \quad A \in C \land \text{Pos} \in C \\
\quad & \quad (N \cup \{(C, t)\}, P, O, t) \sim_w (N, P, O, t) \\
\quad & \quad \{D \in \text{fsts} \cdot (P \cup O) \land \text{subsumes} \ D C \} \\
\quad & \quad (N + \{(C, i)\}, P, O, t) \sim_w (N, P, O, t) \\
\end{align*}
\]
A Verified Prover Based on Ordered Resolution

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

| D ∈ fst ‘N ∧ C ∈ fst ‘P ∧ strictly_subsumes D ⊨ C | D ∈ fst ‘N ∧ C ∈ fst ‘P ∧ strictly_subsumes D ⊨ C
| D ∈ fst ‘N ∧ strictly_subsumes D ⊨ C | D ∈ fst ‘N ∧ strictly_subsumes D ⊨ C
| D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O | D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O
| D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O | D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O
| D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O | D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O
| D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O | D ∈ fst ‘N ∧ C ∈ fst ‘P ⊨ D ⊨ O

where they remove the earliest copy of any clause C ∈ P, they also remove all its copies from P. This property is captured by the following lemma:

**Lemma** preserve_min_P:

\[ S \sim_{w} S' \land (C, i) \in P \land C \in fst 'P \land S \land C \in fst 'P \land S' \land (\forall k. (C, k) \in P \land S \implies k \geq i) \implies (C, i) \in P \land S' \]

This completes our review of RPw. As an intermediate step towards a more concrete prover, we restrict the weight function to be an linear equation that considers both timestamps and clause sizes:

**Locale** weighted_FO_resolution_prover_with_size,

```
fun weight :: 'a wclause ⇒ nat where
  weight (C, i) = size_factor * |C| + timestamp_factor * i
```

where \(|C| = \sum_{A : \text{AEC} \sim \text{AEC}} |A|\). It is easy to prove that this definition of weight is strictly monotone and hence that this locale is a sublocale of weighted_FO_resolution_prover. This gives us a correspondingly specialized version of RPw, that will form the basis of further refinement steps.

The idea of organizing 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 RPw, we must first show that any possible behavior of RPw on states of wstate is a possible behavior of RP on the corresponding values of type state:

**Lemma** weighted_RSPR:

\[ S \sim_{w} S' \implies \text{state_of } S = \text{state_of } S' \]

The proof is by induction on the rules of RPw, 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:

finite D ⇒ finite (inference_between D C)

A binary resolution inference takes two premises, of the form \( CAA = C \lor A_1 \lor \cdots \lor A_k \) and \( DA = \neg A \lor D \), and produces a conclusion \( E = (C \lor D) \lor \sigma \). It can be represented compactly by a tuple of the form \((CAA, DA, AA, A, E)\), where \( AA = A_1 \lor \cdots \lor A_k \). We must show that the set of such
tuples produced by inners_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 $(\mathcal{C}_\mathcal{A}, \mathcal{D}_\mathcal{A}, \mathcal{A}_\mathcal{A}, \mathcal{A}_\mathcal{A})$. Let $\mathcal{D}_\mathcal{C} = \mathcal{D} \cup \{C\}$, and let $n$ be the length of the longest clause in $\mathcal{D}_\mathcal{C}$. Moreover, let $\mathcal{A} = \bigcup_{D \in \mathcal{D}_\mathcal{C}} \text{atms}_D \in \mathcal{D} \cup \mathcal{A}_\mathcal{A} = \mathcal{B} | \text{set_mset} \mathcal{B} \subseteq \mathcal{A} \land |\mathcal{B}| \leq n$. Then all inferences between $\mathcal{D}$ and $\mathcal{C}$ belong to $\mathcal{D}_\mathcal{C} \times \mathcal{D}_\mathcal{C} \times \mathcal{A}_\mathcal{A} \times \mathcal{A}$, a cartesian product of finite sets.

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

**theorem weighted_{RP}_{model}**: 
\[ S \rightsquigarrow_w S' \implies \] 
\[ (I \models \text{grounding}_w S') \iff (I \models \text{grounding}_w S) \] 

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 $\mathcal{N}$ or $\mathcal{P}$ 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 fair_if_finite**: 
\[ \text{finite } S_t \implies \text{fair } (I \models \text{state}_o \of S_t) \]

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

**abbreviation $\text{RP}_{\text{basic}}_{\text{measure}} :: 'a \text{wstate} \Rightarrow \text{nat}^2$** where 
\[ \text{RP}_{\text{basic}}_{\text{measure}} (\mathcal{N}, \mathcal{P}, \lambda, t) \equiv \] 
\[ (\text{sum} (((\lambda(C), |C| + 1) \cdot (\mathcal{N} \cup \mathcal{P} \cup O)), |\mathcal{N}|) \]
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 $\mathcal{N}$. 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 $\mathcal{N}$. Rule 8, which moves a clause from $\mathcal{N}$ to $\mathcal{P}$, 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 weighted_{RP}_{basic}_{measure}_{decreasing}_{\mathcal{N}}**: 
\[ S \rightsquigarrow_w S' \land (C_i) \in \mathcal{N}_{\mathcal{N}} \of S \implies \] 
\[ (\text{RP}_{\text{basic}}_{\text{measure}} S', \text{RP}_{\text{basic}}_{\text{measure}} S) \] 
\[ \in \text{RP}_{\text{basic}}_{\text{rel}} \]

where $\text{RP}_{\text{basic}}_{\text{rel}} = \text{natLess}_w \text{lex} \text{natLess} \text{natLess} = \{(m, n) \mid m < n\}$.

What if a clause $C$ is stuck in $\mathcal{P}$? Let $\text{Lemma preserve_min}_{\mathcal{P}}$ 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 persistent_wclause_in_P_if.persistent_clause**: 
\[ C \in \text{Liminf} (I \models \text{state}_w \of \mathcal{S}_t) \implies \] 
\[ \exists i. (C, i) \in \text{Liminf} (I \models \text{set}_w \of \mathcal{S}_t) \]

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

**abbreviation $\text{RP}_{\text{filtered}}_{\text{measure}} :: (a \text{wclause} \Rightarrow \text{bool}) \Rightarrow 'a \text{wstate} \Rightarrow \text{nat}^2$ where** 
\[ \text{RP}_{\text{filtered}}_{\text{measure}} p (\mathcal{N}, \mathcal{P}, \lambda, t) \equiv \] 
\[ (\text{sum} ((\lambda(C), |C| + 1) \cdot (\mathcal{N} \cup \mathcal{P} \cup O \cup p D i)), |\mathcal{N}|) \]
\[ \text{RP}_{\text{filtered}}_{\text{rel}} = \] 
\[ \text{natLess}_w <\text{lex} <\text{natLess} <\text{natLess} \]

The measure $\text{RP}_{\text{filtered}}_{\text{measure}} (\lambda, i). i \leq w$ 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 $\text{RP}_{\text{basic}}_{\text{measure}} S$ to ensure that the measure decreases when a clause $(C, i)$ such that $i > w$ is simplified. This yields the combined measure:

**abbreviation $\text{RP}_{\text{combined}}_{\text{measure}} :: 'a \text{wstate} \Rightarrow \text{nat}^{3} \times \text{nat}^{3}$ where** 
\[ \text{RP}_{\text{combined}}_{\text{measure}} w S \equiv \] 
\[ (w + 1 - t \of S, \text{RP}_{\text{filtered}}_{\text{measure}} (\lambda, i). i \leq w) S, \text{RP}_{\text{basic}}_{\text{measure}} S) \]
This measure is indeed decreasing with respect to a left-to-right lexicographic order:

**Lemma** \( \text{weighted}_w \text{basic measure decreasing}_P \):

\[
S \sim_w S' \land C_i \in \mathcal{P}_w of \ S \implies \\
(\text{RP combined measure} (\text{weight} C_i) S') \\
\text{RP combined measure} (\text{weight} C_i) S) \\
\in \text{natLess} < \text{RP filtered rel} < \text{lex} > \text{RP basic rel}
\]

By combining the two lemmas \( \text{weighted}_w \text{basic measure decreasing}_N \) and \( \text{weighted}_w \text{basic measure decreasing}_P \), we can prove all derivations starting with \( \mathcal{P} = \mathcal{O} = \emptyset \) fair:

**Theorem** \( \text{weighted}_w \text{fair} : \) fair (\( \text{imap state of } Ss \))

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** \( \text{weighted}_w \text{RP saturated} : \)

\( \text{saturated}_upto (\liminf (\text{imap grounding of } Ss)) \)

**Corollary** \( \text{weighted}_w \text{RP complete} : \)

\( \neg \text{satisfiable} (\text{grounding of } (\text{ld } Ss)) \implies \\
\emptyset \in \mathcal{O}_w of (\liminf (\text{imap state of } Ss)) \)

5 Eliminating Nondeterminism

The third refinement layer defines a functional program \( \text{RP}_3 \) 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:

**Function** \( \text{RP}_3 (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \) is

**repeat forever**

- **if** \( \bot \in \mathcal{P} \cup \mathcal{O} \) **then**
  **return** \( \mathcal{P} \cup \mathcal{O} \)
- **else if** \( \mathcal{N} = \mathcal{P} = \emptyset \) **then**
  **return** \( \mathcal{O} \)
- **else if** \( \mathcal{N} = \emptyset \) **then**
  let \( C \) be a minimal-weight clause in \( \mathcal{P} \);
  \( \mathcal{N} := \text{conclusions of all inferences from } \mathcal{O} \cup \{C\} \)
  involving \( C \), with timestamp \( t \);
  move \( C \) from \( \mathcal{P} \) to \( \mathcal{O} \);
  \( t := t + 1 \)
- **else**
  remove an arbitrary clause \( C \) from \( \mathcal{N} \);
  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 \( \mathcal{N} \) 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** \( 'a del clause = 'a lclause \times \text{nat} \)

**Type Synonym** \( 'a dstate = \\
\text{a dela luse list \times 'a dela luse list \times 'a dela luse list \times nat} \)

The prover is defined inside a locale that inherits \( \text{weighted}_w \text{FO resolution prover with size timestamp factors} \). The core function, \( \text{RP}_d \text{ step} \), performs a single iteration of the main loop. Here is the definition, excluding auxiliary functions:

**Function** \( \text{RP}_d \text{ step} :: 'a dstate \Rightarrow 'a dstate where \)

\( \text{RP}_d \text{ step} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) = \\
\text{if } \exists C_i \in \mathcal{P} \cup \mathcal{O} \text{. fst } C_i = [] \text{ then } \\
([], [], \text{remdup } \mathcal{P} \cup \mathcal{O}, t + |\text{remdup } \mathcal{P}|) \\
\text{else } \mathcal{N} \text{ of } \\
[] \Rightarrow \\
(\text{case } \mathcal{P} \text{ of } \\
[] \Rightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\
| \mathcal{P}_0 \# \mathcal{P}' \Rightarrow \\
\text{let } (C, i) = \text{select_min_weight_clause } \mathcal{P}_0 \mathcal{P}'; \\
\mathcal{N} = \text{map } \lambda D. (D, i)) (\text{remdup} \\
(\text{resolve_rename } C C @ \text{concat} (\text{map} \\
(\text{resolve_rename_both_ways } C \circ \text{fst} \mathcal{O}))); \\
\mathcal{P} = \text{filter } (\lambda(D, i). \text{mset } D @ \text{mset } C) \mathcal{P}; \\
\mathcal{O} = (C, i) \# \mathcal{O}; \\
t = t + 1 \\
in \\
(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\
| (C, i) \# \mathcal{N} \Rightarrow \\
\text{let } C = \text{reduce } (\text{map } \text{fst } (\mathcal{P} \cup \mathcal{O})) [[C] \text{ in } \text{if } C = [] \text{ then } \\
([], [], [[], [], t + 1]) \\
\text{else if } \text{is_tautology } C \\
\vee \text{subsume } (\text{map } \text{fst } (\mathcal{P} \cup \mathcal{O})) C \text{ then } \\
(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\
\text{else } \text{let } \\
\mathcal{P} = \text{reduce_all } C \mathcal{P}; \\
(\text{back_to } \mathcal{P}, \mathcal{O}) = \text{reduce_all2 } C \mathcal{O}; \\
\mathcal{P} = \text{back_to } \mathcal{P} \cup \mathcal{P}_0; \\
\mathcal{O} = \text{filter } ((\neg) \circ \text{strictly_subsume } C \circ \text{fst}) \mathcal{O}; \\
\mathcal{P} = \text{filter } ((\neg) \circ \text{strictly_subsume } C \circ \text{fst}) \mathcal{P}; \\
\mathcal{P} = (C, i) \# \mathcal{P} \\
in \\
(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\
\text{in} \\
\text{append operator.} \)

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
$D$ by another clause $C$ is defined as $\exists x. \ C \cdot x \subseteq D$ (Section 2), where $x$ 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 $x$ using matching, instead of blindly enumerating all possible substitutions.

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

$$\begin{align*}
\text{partial-function (option)} \\
\text{RP}_d :: \to 'a dstate \Rightarrow 'a lclause list option \\
\text{where} \\
\text{RP}_d S = \text{if is_final S then Some (map fst (\mathcal{O}_of S))} \\
\text{else RP}_d (\text{RP}_d_\text{step} S)
\end{align*}$$

Since the recursion may diverge, we cannot introduce the function using the $\text{fun}$ command [22]. Instead, we use $\text{partial-function (option)}$ [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 $\text{RP}_d$-step function to the $\text{RP}_w$ predicate. $\text{RP}_d$-step has a coarser granularity than $\text{RP}_w$: A single invocation on a nonfinal state $S$ can amount to a chain of $\text{RP}_w$ transitions. This is captured by the following (weak-)refinement property:

**lemma nonfinal_deterministic_RP_step:**

$$\neg \text{is_final } S \Rightarrow \text{wstate_of } S \leadsto w \text{wstate_of (RP}_d_\text{step} S)$$

where $\text{wstate_of}$ converts $\text{RP}_d$ states to $\text{RP}_w$ states. The entire proof, including key lemmas, is about 1300 lines long. It follows the case distinctions present in $\text{RP}_d_\text{step}$’s definition:

**case $\exists C \in P @ \mathcal{O}$.** $\text{fst C} = [ ]$:

By induction on $\text{remdup P}$ (where remdup removes duplicates), there must exist a derivation of the form

$$\begin{align*}
\text{wstate_of (N', P, O, t)} \\
\leadsto w_{w_2} \text{wstate_of (_[P, O], t)} \\
\leadsto w_{w_9} \text{wstate_of (N', P', (C, i) \# O, t + 1)} \\
\leadsto w \text{wstate_of ([], [ ], O', t + \text{remdup P'})}
\end{align*}$$

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

**case $N' = [ ]$:**

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

**case $N' = [ ]$:**

It suffices to show that the transition

$$\begin{align*}
\text{wstate_of ([], P, O, t)} \\
\leadsto w_{w_9} \text{wstate_of (N', P', (C, i) \# O, t + 1)}
\end{align*}$$

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

@ $\text{concat (map (resolve_rename_both_ways } C \circ \text{fst}) O))$, and $P' = \text{filter (\lambda (D, j). mset D } \neq \text{ mset C) P}$. The main proof obligation is that $N'$, converted to multisets, equals the multiset $\text{mset_set ((\text{\text{ID. (D, t)})}} \leadsto \text{\text{concl_of } infers_between (set_mset (\text{\text{fst}} O)) C}$ specified in rule $\leadsto w_9$. The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

**otherwise:**

Let $C' = \text{reduce (map fst P \@ map fst O)} [ ]$. If $C' = [ ]$, then

$$\begin{align*}
\text{wstate_of (((C, i) \# N'), P, O, t)} \\
\leadsto w_{w_5} \text{wstate_of ([], [ ], N', P, O, t)} \\
\leadsto w_{w_3} \text{wstate_of ([], [ ], N, P, O, t)} \\
\leadsto w_{w_4} \text{wstate_of ([], [ ], N', P, O, t)} \\
\leadsto w_{w_8} \text{wstate_of (N', ([[], [ ]], [ ], t)} \\
\leadsto w_{w_2} \text{wstate_of ([], [ ], [ ], t)} \\
\leadsto w_{w_9} \text{wstate_of ([], [ ], [ ], t)}
\end{align*}$$

Otherwise, if is_tautology $C' \lor \text{subsume (map fst (P@ O))}$ $C'$, then

$$\begin{align*}
\text{wstate_of (((C, i) \# N'), P, O, t)} \\
\leadsto w_{w_5} \text{wstate_of ((C', i) \# N', P, O, t)} \\
\leadsto w_{w_6} \text{wstate_of ((C', i) \# N', P', O, t)} \\
\leadsto w_{w_7} \text{wstate_of ((C', i) \# N', \text{back_to } P @ P', O', t)} \\
\leadsto w_{w_4} \text{wstate_of ((C', i) \# N', \text{back_to } P @ P', O'', t)} \\
\leadsto w_{w_3} \text{wstate_of ((C', i) \# N', P''', O'', t)} \\
\leadsto w_{w_8} \text{wstate_of (N', (C', i) \# P''', O'', t)}
\end{align*}$$

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

**Soundness and Completeness Proofs.** Let $S_0 = (N_0, [ ], [ ], t_0)$ be an arbitrary initial state. Soundness means that whenever $\text{RP}_d S_0$ terminates with some clause set $R$, then $R$ is a satisfaction that satisfies the same models as $N_0$. Moreover, if $N_0$ is unsatisfiable, then $R$ contains $\bot$, 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 $\text{RP}_w$ to $\text{RP}_d$, we first define $\text{SSS}$ as a full chain of nontrivial $\text{RP}_d$ steps starting from $S_0$. We let $\text{SSS} = \text{derivation_from } S_0$, with $\text{primcorec}$ derivation_from :: $\to 'a dstate \Rightarrow 'a dstate list$

where

$$\text{derivation_from } S = \text{LCons S (if is_final S then LNil else derivation_from (RP_d_step S))}$$

Based on $\text{SSS}$, we let $\text{wSSS} = \text{map wstate_of } \text{SSS}$ and note that $\text{wSSS}$ is a full chain of "big" $\leadsto w_9$ steps. Using a lemma that will be proved below, we obtain a full chain $\text{sswSSS}$ of "small"
The proof is by contradiction:

Assume that \( \neg \) satisfiable (grounding_of \( N_0 \)). Hence, by weighted_RP_complete we have \( 0 \in O_0 \) of sswws. It is easy to show that sswws’s limit is a subset of wss’s limit; hence \( 0 \in O_0 \) of wss. This implies the existence of a natural number \( k \) such that \( 0 \in O_0 \) (lth wss \( k \)). Hence \( 0 \in O_0 \) of (RPd_step \( k \), \( S_0 \)). However, by an induction on \( k \), we can show that RPd must terminate after at most \( k \) iterations, contradicting the assumption that RPd diverges.

A Coinductive Puzzle. A single “big” step of the deterministic prover RPd may correspond to many “small” steps of the weighted prover RPw. To transfer the results from RPw to RPd, 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:

\[
\text{lemma chain_tranclp_imp_exists_chain:}
\begin{align*}
\text{chain } R^+ \text{ xs } \Rightarrow \\
\exists y. \text{ chain } R \ y \wedge \text{ xs } \subseteq \ y \wedge \text{ lhs } \ y = \text{ lhs } \ y \\
\wedge \text{ llast } \ y = \text{ llast } \ y
\end{align*}
\]

where the embedding \( \subseteq \) of lazy lists is defined coinductively using \( ++ \), which preends a finite list to a lazy list:

\[
\text{coinductive } \subseteq : \ a \text{ list } \Rightarrow \ a \text{ list } \Rightarrow \text{ bool } \text{ where}
\begin{align*}
\text{finite } \ y & \Rightarrow \text{ LNil } \subseteq \ y \\
\text{ xs } \subseteq \ y & \Rightarrow \text{ LCons } \ x \ \text{ xs } \subseteq \ y \wedge \text{ LCons } \ x \ \text{ ys}
\end{align*}
\]

fun ++ : a list ++ a list where
\begin{align*}
[] & \Rightarrow \text{ xs } = \text{ xs } \\
\text{ zs } & \Rightarrow \text{ (z } \# \text{ zs } \wedge \text{ xs } = \text{ LCons } \ z \ (\text{ zs } ++ \ \text{ xs })
\end{align*}

The definition of \( \subseteq \) ensures that infinite lazy lists only embed other infinite lazy lists, but not the finite ones: \( \text{ xs } \subseteq \ y \Rightarrow \text{ (finite } \ x \ y \Rightarrow \text{ infinite } \ y) \). 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 coinductively defined witness:

\[
\text{corec wit : } (x \Rightarrow y) \Rightarrow \text{ a list } \Rightarrow \text{ a list } \text{ where}
\begin{align*}
\text{wit } R \ x s = & \ (\text{ case } x s \ of) \\
& \text{ LCons } \ y (\text{ LCons } \ y y s) \Rightarrow \\
& \text{ LCons } \ x \ (\text{ pick } \ x \ y ++ \text{ wit } \ (\text{ LCons } \ y y s)) \\
\text{ _ } & \Rightarrow \text{ xs })
\end{align*}
\]

Here pick \( \ x \ y \) returns an arbitrary finite list of \( R \)-related states connecting the \( R^+ \)-related \( x \) and \( y \). Its definition is pick \( x \ y = (\text{ SOME } \ \text{ zs. chain } R \ (\text{ llist } \ of \ (x \ # \ \text{ zs } @ [y]))) \), where llist_of converts finite lists into lazy lists and SOME is Hilbert’s choice operator. Thus, pick satisfies the characteristic property \( R^+ \ x \ y \Rightarrow \text{ chain } R \ (\text{ llist } \ of \ (x \ # \ \text{ pick } \ x \ y @ [y])) \). 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 \( ++ \), which makes the productivity of this function nontrivial. This syntactic structure of the definition is called corecursive up to \( ++ \). Ultimately, wit is productive because \( ++ \) does not remove any LCons constructors from its second arguments. A slightly weaker requirement, called friendliness, is supported by Isabelle’s corec command [5]. For the above definition to be accepted \( ++ \) must be registered as a “friend.” This involves a one-line proof.

The four conjuncts in chain_tranclp_imp_exists_chain are discharged in turn under the assumption chain \( R^+ \) xs. In order of increasing difficulty: lhd (wit \( R \ x s ) = \text{ lhd } \ y s \) follows by simple rewriting. Next, llast (wit \( R \ x s ) = \text{ llast } \ y s \) requires an induction in the case of finite chains \( xs \). For any infinite chain \( \text{ zs } \), llast \( \text{ zs } \) is defined as a fixed unspecified \( a \) value.

The properties \( \text{ xs } \subseteq \text{ wit } \ x \ y \) and \( \text{ chain } R \ (\text{ wit } \ x s ) \) require a coinduction on \( \subseteq \) and chain, respectively. In keeping with the
where

To simplify notation, in this paper we fix fresh variable names relies on the variables contained in a list of clauses. The creation of IsaFoR list of pairs of constraints. To be executable, the translation easy to translate a finite set of finite sets of terms into a finite

term set set

IsaFoR match:

0
Fun
atm
′

our locales substitution defined as 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 (∃σ. s · σ = t) ∧ (∃σ. 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, ‘ο, and atm_of_atms with ‘. Var, ‘α × ‘β. ‘α × ‘τ, 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 renamings_apart was not present in IsaFoR. We supply a definition:

fun renamings_apart :: term clause list ⇒ (‘u ⇒ term) list

where

renamings_apart [] = []
| renamings_apart (C # Cs) = let as = renamings_apart Cs in
(λu. ‘u + max ((0) (∪ vars_clause_list (Cs · as))) + 1) # as

where vars_clause_list :: term clause list ⇒ ‘u set returns the variables contained in a list of clauses. The creation of fresh variable names relies on ‘u = nat.

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 = (∃K ∈ 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 nonde-terminism 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 clause 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

6 Obtaining Executable Code

Our deterministic prover RP₄ is already quite close to being an executable program. The fourth refinement, the prover RP₅, adds the missing ingredients: a concrete representation of terms and an executable algorithm for clause subsumption.

First-Order Terms. We instantiate our abstract notion of atom using a particularly comprehensive formalization of terms developed as part of the IsaFoR library [51]. This rewriting-independent part of IsaFoR has recently moved to the Archive of Formal Proofs [47].

IsaFoR terms are defined as the following datatype:

datatype (‘f,’v) term = Var ‘v | Fun ‘f ((‘f,’v) term list)

To simplify notation, in this paper we fix ‘f = ‘v = nat and abbreviate (‘f,’v) term by term. In the formalization, polymorphic types are used whenever possible. IsaFoR also defines the standard monadic term substitution :: term ⇒ (‘v ⇒ term) ⇒ term and a unify :: (term × term) list ⇒ lsubst ⇒ lsubst function, where lsubst = (‘v × term) list is the list-based representation of a finite substitution. The function unify computes the MGU for a list of unification 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 (∃σ. s · σ = t) ∧ (∃σ. 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, ‘ο, and atm_of_atms with ‘. Var, ‘α × ‘β. ‘α × ‘τ, 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 renamings_apart was not present in IsaFoR. We supply a definition:

fun renamings_apart :: term clause list ⇒ (‘u ⇒ term) list

where

renamings_apart [] = []
| renamings_apart (C # Cs) = let as = renamings_apart Cs in
(λu. ‘u + max ((0) (∪ vars_clause_list (Cs · as))) + 1) # as

where vars_clause_list :: term clause list ⇒ ‘u set returns the variables contained in a list of clauses. The creation of fresh variable names relies on ‘u = nat.
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) \( \text{leq}_{\text{lit}} \) such that \( \text{leq}_{\text{lit}} L K \) only if \( \text{is\_pos} L = \text{is\_pos} K \) and \( \text{match\_term\_list} \ [\text{atom\_of} L, \text{atom\_of} K] \) \( \sigma \) = Some \( \rho \) for some \( \sigma \) and \( \rho \). Any quasi-order satisfying this property can be used in a refinement of \( \text{subsumes\_list} \) to remove too small literals (with respect to \( \text{leq}_{\text{lit}} \)), as highlighted in gray below:

\[
\text{fun} \ \text{subsumes\_list}' :: \text{term\_literal\_list} \Rightarrow \\
\text{term\_literal\_list} \Rightarrow ('\lor \Rightarrow \text{term\_option}) \Rightarrow \text{bool}
\]

where

\[
\text{subsumes\_list}' [] \ Ks \ \sigma = \text{True} \\
| \text{subsumes\_list}' (L \ # \ Ls) \ Ks \ \sigma = \\
\quad \text{let} \ Ks = \text{filter} \ (\text{leq\_lit} \ L) \ Ks \ \text{in} \\
\quad (\exists K \in \text{set} \ Ks. \ \text{is\_pos} K = \text{is\_pos} L \wedge \\
\quad \quad \text{case} \ \text{match\_term\_list} \ [(\text{atom\_of} L, \text{atom\_of} K)] \ \sigma \ \text{of} \\
\quad \quad \text{None} \Rightarrow \text{False} \\
\quad \quad | \ \text{Some} \ \rho \Rightarrow \text{subsumes\_list}' \ Ls \ (\text{remove1} \ K \ Ks) \ \rho
\]

The lemma \( \text{subsumes\_list} \ Ls \ Ks \ \rho = \text{subsumes\_list}' \ (\text{sort} \ \text{leq\_lit} \ Ls) \ Ks \ \rho \) allows the code generator to refine the original version. In \( \text{RP}_x \), we let \( \text{leq\_lit} \) 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 \( \text{leq\_lit} \). 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

\[
\text{export\_code} \ \text{RP}_x, \ \text{in} \ \text{SML\ module\_name} \ \text{RP}
\]

generates an ML module containing the implementation of our prover in about 1000 lines of code, including dependencies. The generated module exports the function \( \text{RP}_x : (\text{term\_literal\_list \ * \ nat}) \ \text{list} \Rightarrow \text{bool} \). 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 \( \text{RP}_x \) will terminate and return \text{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 \( \text{RP}_x \ t \ = \ b \) 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 \( \text{RP}_x \) with the trusted TPTP parser from Metis [15]. We benchmarked E 2.1, Vampire 4.2.2, Metis 2.4.1, and \( \text{RP}_x \) 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>( \text{RP}_x )</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.

As expected, \( \text{RP}_x \) is not competitive. A prover’s performance comes from its calculus, its heuristics, and its indexing data structures. \( \text{RP}_x \) 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 \( \Phi_2 \) of first-order problems, each consisting of the following \( n + 2 \) clauses:

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

A comment in the benchmark warns us that back in 2007, no prover could solve the \( \Phi_{23} \) within an hour. Even in 2018, only one prover solves \( \Phi_{23} \) within 300 s, and four provers solve \( \Phi_{20} \) within 300 s. \( \text{RP}_x \) solves \( \Phi_{20} \) in 100 s and \( \Phi_{22} \) in 200 s. Presumably, the reason for this success is that \( \text{RP}_x \) fortuitously chooses an instance of the Knuth–Bendix order that is well suited to this benchmark.

\[^3\text{http://matryoshka.gforge.inria.fr/pubs/fun\_rp\_data.tar.gz}\]
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 \texttt{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) 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 Pelletier’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 Pelletier’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 Margerson’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—
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 Summerville, Jørgen Villadsen, and the anonymous reviewers suggested many textual improvements. Eugene Kotelnikov, 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
