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 ordered resolution prover and establish its soundness and refutational completeness. Methodologically, we apply stepwise refinement to obtain, from an abstract specification of a nondeterministic prover, 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.

1 Introduction

Automatic theorem provers based on superposition, such as E [39], SPASS [49], and Vampire [20], are often employed as backends in proof assistants and program verification tools [6, 19, 31]. 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 $L_1 \vee \cdots \vee L_n$ 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., $\mathsf{prime}(n)$. The empty (false) clause is denoted by $\bot$.

Resolution works by refutation: Conceptually, the calculus proves a conjecture $\forall \bar{x}.\, C$ from a set of axioms $\mathcal{D}$ by deriving $\bot$ from $\mathcal{D} \cup \{\exists \bar{x}.\, \neg C\}$, indicating its unsatisfiability. As an optimization, it uses a redundancy criterion to discard tautologies, subsumed clauses, and other unnecessary clauses; for example, $\mathsf{p}(x) \vee \mathsf{q}(x)$ and $\mathsf{p}(5)$ are both subsumed by $\mathsf{p}(x)$. 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 formally verify, using Isabelle/HOL [29], a purely functional prover based on ordered resolution. The verification relies on stepwise refinement [51]. 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 [37, 38]. 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 [10]. 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 [37, 38] 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 [5, 28], 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.

CPP'19, January 14-15, 2019, Cascais/Lisbon, Portugal
2019.

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. The Isabelle files are available in the IsaFoL repository[2] and in the *Archive of Formal Proofs*.[3] 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 [50], 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 [5], which is parameterized by a type $'a$ of logical atoms. Literals L are defined as an inductive datatype: $'a\ literal = \mathrm{Pos}\ 'a \mid \mathrm{Neg}\ 'a$. 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 $\neg A \vee B$, where A and B are arbitrary atoms, is represented by the multiset $\{\mathrm{Neg}\ A, \mathrm{Pos}\ B\}$, and the empty clause $\bot$ is represented by the empty multiset $\emptyset$. The complement operation is defined as $-\mathrm{Neg}\ A = \mathrm{Pos}\ A$ and $-\mathrm{Pos}\ A = \mathrm{Neg}\ A$ 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 $\{y \mapsto x\}$ to the two-literal clause $\mathrm{p}(x) \vee \mathrm{p}(y)$ results in $\mathrm{p}(x) \vee \mathrm{p}(x)$, 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 $\models$ is defined as $I \models A \longleftrightarrow A \in I$. This definition is lifted to literals, clauses, and sets of clauses in the usual way. A set of clauses $\mathcal{D}$ is *satisfiable* if there exists an interpretation I such that $I \models \mathcal{D}$.

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) [47]. 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 the basic operations on substitutions: application ($\cdot$), identity (id), and composition ($\circ$):

locale *substitution_ops* =
  fixes id :: $'s$ **and** $\circ$ :: $'s \Rightarrow 's \Rightarrow 's$ **and** $\cdot$ :: $'a \Rightarrow 's \Rightarrow 'a$

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 $A \longleftrightarrow \forall\sigma.\ A = A \cdot \sigma$. A ground substitution is a substitution whose application always results in ground atoms. Nonstrict and strict generalization are defined as

$$\begin{aligned} \text{generalizes } A\ B &\longleftrightarrow \exists\sigma.\ A \cdot \sigma = B \\ \text{strictly\_generalizes } A\ B &\longleftrightarrow \text{generalizes } A\ B \\ &\quad \wedge \neg\ \text{generalizes } B\ A \end{aligned}$$

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

$$\text{grounding\_of } C = \{C \cdot \sigma \mid \text{is\_ground } \sigma\}$$

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

$$\begin{aligned} \text{subsumes } C\ D &\longleftrightarrow \exists\sigma.\ C \cdot \sigma \subseteq D \\ \text{strictly\_subsumes } C\ D &\longleftrightarrow \text{subsumes } C\ D \\ &\quad \wedge \neg\ \text{subsumes } D\ C \end{aligned}$$

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 :: $'a\ clause\ list \Rightarrow 's\ list$ **and**
    atm_of_atms :: $'a\ list \Rightarrow 'a$
  assumes
    $A \cdot \mathrm{id} = A$
    $A \cdot (\sigma \circ \tau) = (A \cdot \sigma) \cdot \tau$
    $(\forall A.\ A \cdot \sigma = A \cdot \tau) \Longrightarrow \sigma = \tau$
    is_ground $(C \cdot \sigma) \Longrightarrow \exists\tau.$ is_ground $\tau \wedge C \cdot \tau = C \cdot \sigma$
    wf strictly_generalizes
    $|\text{renamings\_apart } Cs| = |Cs|$
    $\rho \in \text{renamings\_apart } Cs \Longrightarrow \text{is\_renaming } \rho$
    var_disjoint $(Cs \cdot \text{renamings\_apart } Cs)$
    atm_of_atms $As \cdot \sigma =$ atm_of_atms $Bs \longleftrightarrow$
      map $(\lambda A.\ A \cdot \sigma)\ As = Bs$

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 :: $'a\ set\ set \Rightarrow 's\ option$ that computes an MGU σ given a set of unification constraints.

[1] https://bitbucket.org/isafol/isafol/wiki/Home
[2] https://bitbucket.org/isafol/isafol/src/master/Functional_Ordered_Resolution_Prover/
[3] https://isa-afp.org/entries/Ordered_Resolution_Prover.html

3 Bachmair and Ganzinger's Prover

Our earlier formalization [37, 38] 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 $\succ$ on nonground atoms so that $B \succ A$ if and only if for all ground substitutions σ, we have $B \cdot \sigma > A \cdot \sigma$. The calculus consists of the single rule

$$\frac{C \vee A_1 \vee \cdots \vee A_k \quad \neg A \vee D}{(C \vee D) \cdot \sigma}$$

where σ is the (canonical) MGU that solves the unification problem $A_1 \stackrel{?}{=} \cdots \stackrel{?}{=} A_k \stackrel{?}{=} A$. In addition, each $A_i \cdot \sigma$ must be strictly $\succ$-maximal with respect to the atoms in $C \cdot \sigma$ (meaning that A_i is not $\preceq$ any atom in $C \cdot \sigma$), and $A \cdot \sigma$ is $\succ$-maximal with respect to the atoms in $D \cdot \sigma$. To achieve completeness, the rule must be adapted slightly to rename apart the variables occurring in different premises.

A set of clauses $\mathcal{D}$ is *saturated* if any conclusion from premises in $\mathcal{D}$ is already in $\mathcal{D}$. The ordered resolution calculus is refutationally complete, meaning that any unsatisfiable saturated set of clauses necessarily contains $\bot$.

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 $\bot$ 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 $\mathcal{D}$ *saturated up to redundancy*, written saturated_upto $\mathcal{D}$, if any inference from nonredundant clauses in $\mathcal{D}$ 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 $\rightsquigarrow$ on states, which are triples $\mathcal{S} = (\mathcal{N}, \mathcal{P}, \mathcal{O})$ of *new clauses* $\mathcal{N}$, *processed clauses* $\mathcal{P}$, and *old clauses* $\mathcal{O}$. Initially, $\mathcal{N}$ is the input problem, and $\mathcal{P} \cup \mathcal{O}$ is empty. Clauses can be removed if they are tautological or subsumed or after subsumption resolution has been applied. When all clauses in $\mathcal{N}$ have been processed (either removed entirely or moved to $\mathcal{P}$), a clause C from $\mathcal{P}$ can be chosen for *inference computation*: C is then moved from $\mathcal{P}$ to $\mathcal{O}$, and all its conclusions with premises from the other old clauses form the new $\mathcal{N}$. Formally:

inductive $\rightsquigarrow$ :: *'a state* $\Rightarrow$ *'a state* $\Rightarrow$ *bool* **where**
  Neg $A \in C \wedge$ Pos $A \in C \Longrightarrow$
  $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_1 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \wedge$ subsumes $D\ C \Longrightarrow$
  $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_2 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \wedge$ strictly_subsumes $D\ C \Longrightarrow$
  $(\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O}) \rightsquigarrow_3 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \wedge$ strictly_subsumes $D\ C \Longrightarrow$
  $(\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C\}) \rightsquigarrow_4 (\mathcal{N}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{P} \cup \mathcal{O} \wedge$ reduces $D\ C\ L \Longrightarrow$
  $(\mathcal{N} \cup \{C \uplus \{L\}\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_5 (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O})$
| $D \in \mathcal{N} \wedge$ reduces $D\ C\ L \Longrightarrow$
  $(\mathcal{N}, \mathcal{P} \cup \{C \uplus \{L\}\}, \mathcal{O}) \rightsquigarrow_6 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $D \in \mathcal{N} \wedge$ reduces $D\ C\ L \Longrightarrow$
  $(\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C \uplus \{L\}\}) \rightsquigarrow_7 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_8 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\emptyset, \mathcal{P} \cup \{C\}, \mathcal{O}) \rightsquigarrow_9$
  (concl_of ' infers_between $\mathcal{O}\ C, \mathcal{P}, \mathcal{O} \cup \{C\}$)

Subscripts on $\rightsquigarrow$ identify the rules. The notation f ' X stands for the image of the (multi)set X under f, infers_between $\mathcal{O}\ C$ calculates all the ordered resolution inferences whose premises are a subset of $\mathcal{O} \cup \{C\}$ that contains C, and reduces $D\ C\ L$ is defined as $\exists D' L' \sigma.\ D = D' \uplus \{L'\} \wedge -L = L' \cdot \sigma \wedge D' \cdot \sigma \subseteq C$.

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

$$\begin{aligned}
&(\{\neg \mathrm{p(a,a)},\ \mathrm{p}(x,x),\ \neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y)\},\ \emptyset,\ \emptyset)\\
\rightsquigarrow_8^+\ &(\emptyset,\ \{\neg \mathrm{p(a,a)},\ \mathrm{p}(x,x),\ \neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y)\},\ \emptyset)\\
\rightsquigarrow_9\ &(\emptyset,\ \{\neg \mathrm{p(a,a)},\ \mathrm{p}(x,x)\},\ \{\neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y)\})\\
\rightsquigarrow_9\ &(\{\mathrm{p}(x,\mathrm{f}(x))\},\ \{\neg \mathrm{p(a,a)}\},\ \{\neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y),\ \mathrm{p}(x,x)\})\\
\rightsquigarrow_8\ &(\emptyset,\ \{\neg \mathrm{p(a,a)},\ \mathrm{p}(x,\mathrm{f}(x))\},\ \{\neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y),\ \mathrm{p}(x,x)\})\\
\rightsquigarrow_9\ &(\{\mathrm{p}(x,\mathrm{f(f}(x)))\},\ \{\neg \mathrm{p(a,a)}\},\\
&\quad \{\neg \mathrm{p(f}(x),y) \vee \mathrm{p}(x,y),\ \mathrm{p}(x,x),\ \mathrm{p}(x,\mathrm{f}(x))\})\\
\rightsquigarrow_8\ &\cdots
\end{aligned}$$

We can leave $\neg \mathrm{p(a,a)}$ in $\mathcal{P}$ forever and always generate more clauses of the form $\mathrm{p}(x, \mathrm{f}^i(x))$, for increasing values of i. This emphasizes the importance of a fair clause selection strategy.

Formally, a *derivation* is a possibly infinite sequence of states $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \mathcal{S}_2 \rightsquigarrow \cdots$. 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 $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \cdots$ formally as chain $(\rightsquigarrow)$ $\mathcal{S}s$, where $\mathcal{S}s$ is a lazy list of states and chain is a coinductive predicate:

coinductive chain :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow$ *'a llist* $\Rightarrow$ *bool*
where
  chain R (LCons x LNil)
| chain $R\ xs \wedge R\ x$ (lhd xs) $\Longrightarrow$ chain R (LCons $x\ xs$)

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 an at most finite prefix:

definition Liminf :: *'a set llist* ⇒ *'a set* **where**
Liminf *Xs* = $\bigcup_{i<|Xs|} \bigcap_{j:i\le j<|Xs|}$ 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 Liminf $\mathcal{S}s$ = (Liminf $\mathcal{N}s$, Liminf $\mathcal{P}s$, Liminf $\mathcal{O}s$), where $\mathcal{N}s$, $\mathcal{P}s$, and $\mathcal{O}s$ are the projections of the $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$ components of $\mathcal{S}s$.

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

theorem *RP_sound*:
$\emptyset \in$ clss_of (Liminf $\mathcal{S}s$) ⟹
¬ satisfiable (grounding_of (lhd $\mathcal{S}s$))

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

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

theorem *RP_model*:
$\mathcal{S} \rightsquigarrow \mathcal{S}'$ ⟹
($I \models$ grounding_of $\mathcal{S}'$ ⟷ $I \models$ grounding_of $\mathcal{S}$)

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 $\mathcal{N}$ or $\mathcal{P}$. Accordingly, fairness is defined as Liminf $\mathcal{N}s$ = Liminf $\mathcal{P}s = \emptyset$. The completeness theorem states that the limit of a fair derivation $\mathcal{S}s$ is saturated:

theorem *RP_saturated_if_fair*:
fair $\mathcal{S}s$ ⟹ saturated_upto (Liminf (grounding_of $\mathcal{S}s$))

In particular, if the initial problem is unsatisfiable, $\bot$ must appear in the $\mathcal{O}$ component of the limit of any fair derivation:

corollary *RP_complete_if_fair*:
fair $\mathcal{S}s$ ∧ ¬ satisfiable (grounding_of (lhd $\mathcal{S}s$)) ⟹
$\emptyset \in$ $\mathcal{O}$_of (Liminf $\mathcal{S}s$)

4 Ensuring Fairness

The second refinement layer is the prover RP_w, which ensures fairness by assigning a *weight* to every clause and by organizing the set of processed clauses—the $\mathcal{P}$ state component—as a priority queue, where lighter clauses are chosen before heavier clauses. 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 full_chain $(\rightsquigarrow_\text{w})$ $\mathcal{S}s$. For the rest of this section, we fix a full chain $\mathcal{S}s$ such that $\mathcal{P}$_of (lhd $\mathcal{S}s$) = $\mathcal{O}$_of (lhd $\mathcal{S}s$) = $\emptyset$.

Because each RP_w rule corresponds to an RP rule, it is straightforward to lift the soundness and completeness results from RP to 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 RP_w, without the assumption fair $\mathcal{S}s$.

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 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 $\mathcal{S} = (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$, 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:

type_synonym *'a wclause* = *'a clause* × *nat*
type_synonym *'a wstate* =
'a wclause multiset × *'a wclause multiset*
× *'a wclause multiset* × *nat*

We extend the *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:

locale *weighted_FO_resolution_prover* =
FO_resolution_prover +
fixes weight :: *'a wclause* ⇒ *nat*
assumes $i < j$ ⟹ weight (C, i) < 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 [40].

The RP_w prover uses *'a wclause* for clauses. It is defined inductively as follows:

inductive $\rightsquigarrow_\text{w}$ :: *'a wstate* ⇒ *'a wstate* ⇒ *bool* **where**
Neg $A \in C \wedge$ Pos $A \in C$ ⟹
$(\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_\text{w1} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
| $D \in$ fst ` $(\mathcal{P} \uplus \mathcal{O}) \wedge$ subsumes $D\ C$ ⟹
$(\mathcal{N} + \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_\text{w2} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
| $D \in$ fst ` $\mathcal{N} \wedge C \in$ fst ` $\mathcal{P} \wedge$ strictly_subsumes $D\ C$ ⟹
$(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_\text{w3} (\mathcal{N}, \{(E, k) \in \mathcal{P}.\ E \neq C\}, \mathcal{O}, t)$
| $D \in$ fst ` $\mathcal{N} \wedge$ strictly_subsumes $D\ C$ ⟹
$(\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C, i)\}, t) \rightsquigarrow_\text{w4} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
| $D \in$ fst ` $(\mathcal{P} \uplus \mathcal{O}) \wedge$ reduces $D\ C\ L$ ⟹
$(\mathcal{N} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_\text{w5} (\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t)$
| $D \in$ fst ` $\mathcal{N} \wedge$ reduces $D\ C\ L$
$\wedge\ (\forall j.\ (C \uplus \{L\}, j) \in \mathcal{P} \Longrightarrow j \le i)$ ⟹
$(\mathcal{N}, \mathcal{P} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{O}, t) \rightsquigarrow_\text{w6} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
| $D \in$ fst ` $\mathcal{N} \wedge$ reduces $D\ C\ L$ ⟹
$(\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C \uplus \{L\}, i)\}, t) \rightsquigarrow_\text{w7} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
| $(\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_\text{w8} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
| $(\forall (D, j) \in \mathcal{P}.$ weight $(C, i) \le$ weight $(D, j)) \wedge$
$\mathcal{N}$ = mset_set $((\lambda D.\ (D, t))$ ` concl_of ` infers_between
(set_mset (fst ` $\mathcal{O}$)) C) ⟹
$(\emptyset, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t) \rightsquigarrow_\text{w9}$
$(\mathcal{N}, \{(D, j) \in \mathcal{P}.\ D \neq C\}, \mathcal{O} \uplus \{(C, i)\}, t + 1)$

where fst 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.

RP_w uses finite multisets for representing $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$. They offer a compromise between the layer 1 representation as sets and the layer 3 implementation as lists. Finite multisets also help eliminate some unfair derivations. The finiteness condition guarantees that each clause in $\mathcal{N}$ gets the opportunity to move to $\mathcal{P}$ (and further to $\mathcal{O}$). Moreover, whereas the set-based RP allows idle transitions, such as $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$ for $C \in \mathcal{N} \cap \mathcal{P}$, the use of multisets and $\uplus$ precludes such transitions in RP_w.

The last transition rule, which computes inferences, assigns timestamp t to each newly computed clause D and increments t. Since we want $\mathcal{P}$ to work as a priority queue, RP_w chooses a clause C with the smallest weight.

Timestamps are preserved when clauses are moved between $\mathcal{N}$, $\mathcal{P}$, 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 a 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

$$\begin{aligned} &(\_, \_ \uplus \{(C,0)\}, \_) \rightsquigarrow_\text{w} \\ &(\_, \_ \uplus \{(C,0),(C,1)\}, \_) \rightsquigarrow_\text{w} (\_, \_ \uplus \{(C,1)\}, \_) \rightsquigarrow_\text{w}^+ \\ &(\_, \_ \uplus \{(C,1),(C,2)\}, \_) \rightsquigarrow_\text{w} (\_, \_ \uplus \{(C,2)\}, \_) \rightsquigarrow_\text{w}^+ \cdots \end{aligned}$$

To prevent this, the RP_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*:
$\mathcal{S} \rightsquigarrow_\text{w} \mathcal{S}' \wedge (C, i) \in \mathcal{P}\_\text{of}\ \mathcal{S} \wedge C \in \text{fst}\ `\ \mathcal{P}\_\text{of}\ \mathcal{S}'$
$\wedge\ (\forall k.\ (C, k) \in \mathcal{P}\_\text{of}\ \mathcal{S} \Longrightarrow k \geq i) \Longrightarrow$
$(C, i) \in \mathcal{P}\_\text{of}\ \mathcal{S}'$

This completes our review of RP_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_timestamp_factors* =
FO_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 $|C| = \sum_{A:\, A \in C \vee \neg A \in C} |A|$. 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 RP_w that 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 RP_w, we must first show that any possible behavior of RP_w on states of type *wstate* is a possible behavior of RP on the corresponding values of type *state*:

lemma *weighted_RP_imp_RP*:
$\mathcal{S} \rightsquigarrow_\text{w} \mathcal{S}' \Longrightarrow$ state_of $\mathcal{S} \rightsquigarrow$ state_of $\mathcal{S}'$

The proof is by induction on the rules of RP_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*:
finite $\mathcal{D} \Longrightarrow$ finite (infers_between $\mathcal{D}$ C)

A binary resolution inference takes two premises, of the form $CAA = C \vee A_1 \vee \cdots \vee A_k$ and $DA = \neg A \vee D$, and produces a conclusion $E = (C \vee D) \cdot \sigma$. It can be represented compactly by a tuple of the form (CAA, DA, AA, A, E), where $AA = A_1 \vee \cdots \vee A_k$. We must show that the set of such tuples produced by infers_between is finite, assuming $\mathcal{D}$ is finite. First, observe that the last component E of a tuple is determined by the other four. Hence it suffices to consider quadruples (CAA, DA, AA, A). Let $\mathcal{DC} = \mathcal{D} \cup \{C\}$, and let n be the length of the longest clause in $\mathcal{DC}$. Moreover, let $\mathcal{A} = \bigcup_{D \in \mathcal{DC}}$ atms_of D and $\mathcal{AA} = \{\mathcal{B} \mid$ set_mset $\mathcal{B} \subseteq \mathcal{A} \wedge |\mathcal{B}| \leq n\}$. Then all inferences between $\mathcal{D}$ and C belong to $\mathcal{DC} \times \mathcal{DC} \times \mathcal{AA} \times \mathcal{A}$, a cartesian product of finite sets.

Soundness and Completeness Proofs. Using the refinement lemma *weighted_RP_imp_RP*, it is easy to lift the *RP_model* theorem (Section 3) to RP_w:

theorem *weighted_RP_model*:
$\mathcal{S} \rightsquigarrow_\text{w} \mathcal{S}' \Longrightarrow$
$(I \models$ grounding_of $\mathcal{S}' \Longleftrightarrow I \models$ grounding_of $\mathcal{S})$

Completeness is considerably more difficult. We first show that the use of timestamps ensures that all full 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*:
lfinite $\mathcal{S}s \Longrightarrow$ fair (lmap state_of $\mathcal{S}s$)

There are two ways in which an infinite derivation $\mathcal{S}s$ in 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 $>$ on *nat* to pairs:

abbreviation RP_basic_measure :: *'a wstate* $\Rightarrow$ nat^2 **where**
RP_basic_measure $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \equiv$
$(\mathrm{sum}\ ((\lambda(C, \_).\ |C| + 1)\ `\ (\mathcal{N} \uplus \mathcal{P} \uplus \mathcal{O})),\ |\mathcal{N}|)$

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_N*:
$\mathcal{S} \rightsquigarrow_\mathrm{w} \mathcal{S}' \wedge (C, \_) \in \mathcal{N}\_\mathrm{of}\ \mathcal{S} \Longrightarrow$
(RP_basic_measure $\mathcal{S}'$, RP_basic_measure $\mathcal{S}$)
$\in$ RP_basic_rel

where RP_basic_rel = natLess <lex> natLess and natLess = $\{(m, n) \mid m < n\}$.

What if a clause C is stuck in $\mathcal{P}$? Lemma *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 *persistent_wclause_in_P_if_persistent_clause*:
$C \in$ Liminf (lmap $\mathcal{P}$_of (lmap state_of $\mathcal{S}s$)) $\Longrightarrow$
$\exists i.\ (C, i) \in$ Liminf (lmap (set_mset $\circ$ $\mathcal{P}$_of) $\mathcal{S}s$)

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

abbreviation RP_filtered_measure ::
('a wclause $\Rightarrow$ *bool)* $\Rightarrow$ *'a wstate* $\Rightarrow$ nat^3 **where**
RP_filtered_measure p $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \equiv$
$(\mathrm{sum}\ ((\lambda(C, \_).\ |C| + 1)\ `\ \{Di \in \mathcal{N} \uplus \mathcal{P} \uplus \mathcal{O} \mid p\ Di\}),$
$|\{Di \in \mathcal{N} \mid p\ Di\}|,\ |\{Di \in \mathcal{P} \mid p\ Di\}|)$

Notice that the case RP_filtered_measure $(\lambda\_.$ True) essentially amounts to RP_basic_measure. In the formalization, we use RP_filtered_measure $(\lambda\_.$ True) to avoid duplication.

Suppose the clause C that is stuck in $\mathcal{P}$ has weight w in the limit, and suppose that a clause D is moved from $\mathcal{P}$ to $\mathcal{O}$ 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: $(\rightsquigarrow^*_{\mathrm{w}1-8} \circ \rightsquigarrow_{\mathrm{w}9})^\omega$. 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 RP_filtered_rel :: $(nat^3)^2$ *set* **where**
RP_filtered_rel $\equiv$
natLess <lex> natLess <lex> natLess

The measure 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 RP_basic_measure $\mathcal{S}$ to ensure that the measure decreases when a clause (C, i) such that $i > w$ is simplified. This yields the combined measure

abbreviation RP_combined_measure ::
$nat \Rightarrow$ *'a wstate* $\Rightarrow$ $nat \times nat^3 \times nat^3$ **where**
RP_combined_measure w $\mathcal{S}$ $\equiv$
$(w + 1 -$ t_of $\mathcal{S}$,
RP_filtered_measure $(\lambda(\_, i).\ i \leq w)$ $\mathcal{S}$,
RP_basic_measure $\mathcal{S})$

This measure is indeed decreasing with respect to a left-to-right lexicographic order:

lemma *weighted_RP_basic_measure_decreasing_P*:
$\mathcal{S} \rightsquigarrow_\mathrm{w} \mathcal{S}' \wedge Ci \in \mathcal{P}\_\mathrm{of}\ \mathcal{S} \Longrightarrow$
(RP_combined_measure (weight Ci) $\mathcal{S}'$,
RP_combined_measure (weight Ci) $\mathcal{S}$)
$\in$ natLess <lex> RP_filtered_rel <lex> 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 (lmap state_of $\mathcal{S}s$)

Since all derivations are fair and RP_w derivations correspond to RP derivations, it is trivial to lift RP's saturation and completeness theorems:

corollary *weighted_RP_saturated*:
saturated_upto (Liminf (lmap grounding_of $\mathcal{S}s$))

corollary *weighted_RP_complete*:
$\neg$ satisfiable (grounding_of (lhd $\mathcal{S}s$)) $\Longrightarrow$
$\emptyset \in \mathcal{O}\_\mathrm{of}$ (Liminf (lmap state_of $\mathcal{S}s$))

5 Eliminating Nondeterminism

The third refinement layer defines a functional program $\mathrm{RP_d}$ that embodies a specific rule application strategy, thereby eliminating $\mathrm{RP_w}$'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 $\mathrm{RP_d}\,(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ **is**
  repeat forever
    if $\bot \in \mathcal{P} \uplus \mathcal{O}$ **then**
      return $\mathcal{P} \uplus \mathcal{O}$
    else if $N = P = \emptyset$ **then**
      return $\mathcal{O}$
    else if $N = \emptyset$ **then**
      let C be a minimal-weight clause in $\mathcal{P}$;
      $\mathcal{N}$:= conclusions of all inferences from $\mathcal{O} \uplus \{C\}$
          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 $\mathcal{P} \uplus \mathcal{O}$;
      if $C = \bot$ **then**
        return $\{\bot\}$
      else if C is neither a tautology not subsumed by
          a clause in $\mathcal{P} \uplus \mathcal{O}$ **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 [48].

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, $\mathrm{RP_d}$_step, performs a single iteration of the main loop. Here is the definition, excluding auxiliary functions:

fun $\mathrm{RP_d}$_step :: *'a dstate* ⇒ *'a dstate* **where**
  $\mathrm{RP_d}$_step $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ =
  if $\exists Ci \in \mathcal{P} @ \mathcal{O}$. fst Ci = [] then
    ([], [], remdups $\mathcal{P}$ @ $\mathcal{O}$, t + |remdups $\mathcal{P}$|)
  else
    (case $\mathcal{N}$ of
      [] ⇒
        (case $\mathcal{P}$ of
          [] ⇒ $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
        | P_0 # $\mathcal{P}'$ ⇒
          let
            (C, i) = select_min_weight_clause P_0 $\mathcal{P}'$;
            $\mathcal{N}$ = map (λD. (D, t)) (remdups
              (resolve_rename C C @ concat (map
                (resolve_rename_both_ways C ∘ fst) $\mathcal{O}$)));
            $\mathcal{P}$ = filter ($\lambda(D, j)$. mset D ≠ mset C) $\mathcal{P}$;
            $\mathcal{O}$ = (C, i) # $\mathcal{O}$;
            t = t + 1
          in
            $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$)
      | (C, i) # $\mathcal{N}$ ⇒
        let
          C = reduce (map fst ($\mathcal{P}$ @ $\mathcal{O}$)) [] C
        in
          if C = [] then
            ([], [], [([], i)], t + 1)
          else if is_tautology C
              ∨ subsume (map fst ($\mathcal{P}$ @ $\mathcal{O}$)) C then
            $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
          else let
            $\mathcal{P}$ = reduce_all C $\mathcal{P}$;
            (back_to_$\mathcal{P}$, $\mathcal{O}$) = reduce_all2 C $\mathcal{O}$;
            $\mathcal{P}$ = back_to_$\mathcal{P}$ @ $\mathcal{P}$;
            $\mathcal{O}$ = filter ((¬) ∘ strictly_subsume C ∘ fst) $\mathcal{O}$;
            $\mathcal{P}$ = filter ((¬) ∘ strictly_subsume C ∘ fst) $\mathcal{P}$;
            $\mathcal{P}$ = (C, i) # $\mathcal{P}$
          in
            $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$)

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

The code relies on some nonexecutable constructs. 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 $\exists\sigma.\ C{\cdot}\sigma \subseteq D$ (Section 2), where σ 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 subsumption can compute a witness σ using matching, instead of blindly enumerating all possible substitutions.

The prover's main program is a tail-recursive function that repeatedly calls $\mathrm{RP_d}$_step until a final state $([], [], \mathcal{O}, t)$ is reached, at which point it returns the clause set $\mathcal{O}$ stripped of its timestamps:

partial_function (*option*)
  $\mathrm{RP_d}$ :: *'a dstate* ⇒ *'a lclause list option*
where
  $\mathrm{RP_d}$ $\mathcal{S}$ = if is_final $\mathcal{S}$ then Some (map fst ($\mathcal{O}$_of $\mathcal{S}$))
      else $\mathrm{RP_d}$ ($\mathrm{RP_d}$_step $\mathcal{S}$)

Since the recursion is not guaranteed to terminate, we cannot introduce the function using the **fun** command [21]. Instead, we use **partial_function** (*option*) [22], which puts the computation in an option monad. The function's result is of the form Some R if the recursion terminates and None if the computation diverges. Executing the function would never actually return None, but it is convenient to define it mathematically in this way. For example, it allows us to state and prove a characterization such as the following, which can be used to replace a terminating call $\mathsf{RP_d}\ \mathcal{S}$ by a finite iteration $\mathsf{RP_d\_step}^k\ \mathcal{S}$:

lemma *deterministic_RP_SomeD*:
$\mathsf{RP_d}\ \mathcal{S} = \mathsf{Some}\ R \Longrightarrow$
$\exists \mathcal{S}'\ k.\ \mathsf{RP_d\_step}^k\ \mathcal{S} = \mathcal{S}' \wedge \mathsf{is\_final}\ \mathcal{S}'$
$\wedge\ R = \mathsf{map\ fst}\ (\mathcal{O}\_\mathsf{of}\ \mathcal{S}')$

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

lemma *nonfinal_deterministic_RP_step*:
$\neg\ \mathsf{is\_final}\ \mathcal{S} \Longrightarrow$
$\mathsf{wstate\_of}\ \mathcal{S} \rightsquigarrow_{\mathsf{w}}^{+} \mathsf{wstate\_of}\ (\mathsf{RP_d\_step}\ \mathcal{S})$

where wstate_of converts $\mathsf{RP_d}$ states to $\mathsf{RP_w}$ states. The entire proof, including key lemmas, is about 1300 lines long. It follows the case distinctions present in $\mathsf{RP_d\_step}$'s definition:

case $\exists Ci \in \mathcal{P} @ \mathcal{O}.\ \mathsf{fst}\ Ci = []$:
By induction on $|\mathsf{remdups}\ \mathcal{P}|$ (where remdups removes duplicates), there must exist a derivation of the form

$$\begin{aligned} & \mathsf{wstate\_of}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}2}^{*}\ & \mathsf{wstate\_of}\ ([], \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}9}\ & \mathsf{wstate\_of}\ (\mathcal{N}', \mathcal{P}', (C, i) \# \mathcal{O}, t + 1) \\ \rightsquigarrow_{\mathsf{w}}^{*}\ & \mathsf{wstate\_of}\ ([], [], \mathcal{O}', t + |\mathsf{remdups}\ \mathcal{P}'|) \end{aligned}$$

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

case $\mathcal{N} = \mathcal{P} = []$:
Contradiction with the assumption that $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ is a nonfinal state.

case $\mathcal{N} = []$:
It suffices to show that the transition

$$\begin{aligned} & \mathsf{wstate\_of}\ ([], \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}9}\ & \mathsf{wstate\_of}\ (\mathcal{N}', \mathcal{P}', (C, i) \# \mathcal{O}, t + 1) \end{aligned}$$

is possible, where $(C, i) \in \mathcal{P}$ is a minimal-weight clause, $\mathcal{N}' = \mathsf{map}\ (\lambda D.\ (D, t))\ (\mathsf{remdups}\ (\mathsf{resolve\_rename}\ C\ C\ @\ \mathsf{concat}\ (\mathsf{map}\ (\mathsf{resolve\_rename\_both\_ways}\ C \circ \mathsf{fst})\ \mathcal{O})))$, and $\mathcal{P}' = \mathsf{filter}\ (\lambda(D, j).\ \mathsf{mset}\ D \neq \mathsf{mset}\ C)\ \mathcal{P}$. The main proof obligation is that $\mathcal{N}'$, converted to multisets, equals the multiset $\mathsf{mset\_set}\ ((\lambda D.\ (D, t))\ `\ \mathsf{concl\_of}\ `\ \mathsf{infers\_between}\ (\mathsf{set\_mset}\ (\mathsf{fst}\ `\ \mathcal{O}))\ C)$ specified in rule $\rightsquigarrow_{\mathsf{w}9}$. The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

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

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}5}^{*}\ & \mathsf{wstate\_of}\ (([], i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}3}^{*}\ & \mathsf{wstate\_of}\ (([], i) \# \mathcal{N}', [], \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}4}^{*}\ & \mathsf{wstate\_of}\ (([], i) \# \mathcal{N}', [], [], t) \\ \rightsquigarrow_{\mathsf{w}8}\ & \mathsf{wstate\_of}\ (\mathcal{N}', [([], i)], [], t) \\ \rightsquigarrow_{\mathsf{w}2}^{*}\ & \mathsf{wstate\_of}\ ([], [([], i)], [], t) \\ \rightsquigarrow_{\mathsf{w}9}\ & \mathsf{wstate\_of}\ ([], [], [([], i)], t) \end{aligned}$$

Otherwise, if $\mathsf{is\_tautology}\ C' \vee \mathsf{subsume}\ (\mathsf{map\ fst}\ (\mathcal{P} @ \mathcal{O}))\ C'$, then

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \# \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}5}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}1,2}\ & \mathsf{wstate\_of}\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \end{aligned}$$

Otherwise:

$$\begin{aligned} & \mathsf{wstate\_of}\ ((C, i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}5}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}6}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}', \mathcal{P}', \mathcal{O}, t) \\ \rightsquigarrow_{\mathsf{w}7}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}', \mathsf{back\_to}\_\mathcal{P} @ \mathcal{P}', \mathcal{O}', t) \\ \rightsquigarrow_{\mathsf{w}4}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}', \mathsf{back\_to}\_\mathcal{P} @ \mathcal{P}', \mathcal{O}'', t) \\ \rightsquigarrow_{\mathsf{w}3}^{*}\ & \mathsf{wstate\_of}\ ((C', i) \# \mathcal{N}', \mathcal{P}'', \mathcal{O}'', t) \\ \rightsquigarrow_{\mathsf{w}8}\ & \mathsf{wstate\_of}\ (\mathcal{N}', (C', i) \# \mathcal{P}'', \mathcal{O}'', t) \end{aligned}$$

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

Soundness and Completeness Proofs. Let $\mathcal{S}_0 = (\mathcal{N}_0, [], [], t_0)$ be an arbitrary initial state. For $\mathsf{RP_d}$, soundness means that whenever $\mathsf{RP_d}\ \mathcal{S}_0$ terminates with some clause set R, then R is a saturation that satisfies the same models as $\mathcal{N}_0$. In addition, if $\mathcal{N}_0$ is unsatisfiable, then R contains $\bot$, which provides a simple syntactic check for unsatisfiability. Completeness means that divergence is possible only if $\mathcal{N}_0$ is satisfiable. For satisfiable clause sets $\mathcal{N}_0$, both termination and divergence are possible.

To lift soundness and completeness results from $\mathsf{RP_w}$ to $\mathsf{RP_d}$, we first define $\mathcal{S}s$ as a full chain of nontrivial $\mathsf{RP_d}$ steps starting from $\mathcal{S}_0$. We let $\mathcal{S}s = \mathsf{derivation\_from}\ \mathcal{S}_0$, with

primcorec derivation_from :: *'a dstate* ⇒ *'a dstate llist* **where**
derivation_from $\mathcal{S}$ = LCons $\mathcal{S}$ (if is_final $\mathcal{S}$ then LNil
else derivation_from ($\mathsf{RP_d\_step}\ \mathcal{S}$))

Based on $\mathcal{S}s$, we let $w\mathcal{S}s = \mathsf{lmap\ wstate\_of}\ \mathcal{S}s$ and note that $w\mathcal{S}s$ is a full chain of "big" $\rightsquigarrow_{\mathsf{w}}^{+}$ steps. Using a lemma that will be proved below, we obtain a full chain $ssw\mathcal{S}s$ of "small" $\rightsquigarrow_{\mathsf{w}}$ steps. This chain satisfies the conditions postulated on $\mathcal{S}s$ in Section 4, allowing us to lift the results presented there.

The soundness results are proved in a nameless locale, or *context*, that assumes termination of $\mathrm{RP_d}$:

fixes $R :: 'a\ lclause\ list$
assumes $\mathrm{RP_d}\ \mathcal{S}_0 = \mathrm{Some}\ R$

The definition of $\mathrm{RP_d}$, using **partial_function**, gives us an induction rule restricted to the case where $\mathrm{RP_d}$ terminates (i.e., returns a Some value). This rule can be used to prove that $\mathcal{S}s$ and hence $w\mathcal{S}s$ and $ssw\mathcal{S}s$ are finite sequences.

Soundness takes the form of a pair of theorems that lift *weighted_RP_model* and *weighted_RP_saturated*:

theorem *deterministic_RP_model*:
$I \models$ grounding_of $\mathcal{N}_0 \longleftrightarrow I \models$ grounding_of R

theorem *deterministic_RP_saturated*:
saturated_upto (grounding_of R)

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

corollary *deterministic_RP_refutation*:
$\neg$ satisfiable (grounding_of $\mathcal{N}_0$) $\longleftrightarrow \emptyset \in R$

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

theorem *deterministic_RP_complete*:
satisfiable (grounding_of $\mathcal{N}_0$)

The proof is by contradiction:

Assume that $\neg$ satisfiable (grounded_of $\mathcal{N}_0$). Hence, by *weighted_RP_complete* we have $\emptyset \in \mathcal{O}$_of $ssw\mathcal{S}s$. It is easy to show that $ssw\mathcal{S}s$'s limit is a subset of $w\mathcal{S}s$'s limit; hence $\emptyset \in \mathcal{O}$_of $w\mathcal{S}s$. This implies the existence of a natural number k such that $\emptyset \in \mathcal{O}$_of (lnth $w\mathcal{S}s\ k$). Hence $\emptyset \in \mathcal{O}$_of ($\mathrm{RP_d}$_step$^k\ \mathcal{S}_0$). However, by an induction on k, we can show that $\mathrm{RP_d}$ must terminate after at most k iterations, contradicting the assumption that $\mathrm{RP_d}$ diverges.

A Coinductive Puzzle. A single "big" step of the deterministic prover $\mathrm{RP_d}$ may correspond to many "small" steps of the weighted prover $\mathrm{RP_w}$. To transfer the results from $\mathrm{RP_w}$ to $\mathrm{RP_d}$, 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:

lemma *chain_tranclp_imp_exists_chain*:
chain $R^+\ xs \Longrightarrow$
$\exists ys.$ chain $R\ ys \wedge xs \sqsubseteq ys \wedge$ lhd $xs =$ lhd ys
$\wedge$ llast $xs =$ llast ys

where the embedding $\sqsubseteq$ of lazy lists is defined coinductively using $+\!\!+$, which prepends a finite list to a lazy list:

coinductive $\sqsubseteq :: 'a\ llist \Rightarrow 'a\ llist \Rightarrow bool$ **where**
lfinite $xs \Longrightarrow$ LNil $\sqsubseteq xs$
| $xs \sqsubseteq ys \Longrightarrow$ LCons $x\ xs \sqsubseteq zs +\!\!+$ LCons $x\ ys$

fun $+\!\!+ :: 'a\ list \Rightarrow 'a\ llist \Rightarrow 'a\ llist$ **where**
$[] +\!\!+ xs = xs$
| $(z \# zs) +\!\!+ xs =$ LCons $z\ (zs +\!\!+ xs)$

The definition of $\sqsubseteq$ ensures that infinite lazy lists only embed other infinite lazy lists, but not the finite ones: $xs \sqsubseteq ys \Longrightarrow$ (lfinite $xs \longleftrightarrow$ lfinite 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 corecursively defined witness:

corec wit :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ llist \Rightarrow 'a\ llist$
where
wit $R\ xs$ = (case xs of
LCons x (LCons $y\ ys$) $\Rightarrow$
LCons x (pick $R\ x\ y +\!\!+$ wit R (LCons $y\ ys$))
| _ $\Rightarrow xs$)

Here pick $R\ x\ y$ returns an arbitrary finite list of R-related states connecting the R^+-related x and y. Its definition is pick $R\ x\ y$ = (SOME zs. chain R (llist_of ($x \# zs$ @ [y]))), where llist_of converts finite lists into lazy lists and SOME is Hilbert's choice operator. Thus, pick satisfies the characteristic property $R^+\ x\ y \Longrightarrow$ chain R (llist_of ($x \#$ pick $R\ x\ y$ @ [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 [4]. 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 $R\ xs$) = lhd xs follows by simple rewriting. Next, llast (wit $R\ xs$) = llast xs requires an induction in the case of finite chains xs. For any infinite chain zs, llast zs is defined as a fixed unspecified $'a$ value. The properties $xs \sqsubseteq$ wit $R\ xs$ and chain R (wit $R\ xs$) require a coinduction on $\sqsubseteq$ and chain, respectively. In keeping with the definition, plain coinduction on $\sqsubseteq$ and chain does not suffice, and we must use coinduction up to $+\!\!+$ on $\sqsubseteq$ and chain.

The property *chain_tranclp_imp_exists_chain* easily extends to full chains.

6 Obtaining Executable Code

Our deterministic prover RP_d is already quite close to being an executable program. The fourth refinement, the prover RP_x, 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 [47]. This rewriting-independent part of IsaFoR has recently moved to the *Archive of Formal Proofs* [44].

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 $\cdot :: term \Rightarrow ('v \Rightarrow term) \Rightarrow term$ and a unify $:: (term \times term)\ list \Rightarrow lsubst \Rightarrow lsubst$ function, where $lsubst = ('v \times term)\ list$ 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 $(\exists\sigma.\ s \cdot \sigma = t) \wedge (\nexists\sigma.\ t \cdot \sigma = s)$.

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 $'v \Rightarrow term$ and the constants $\cdot$, id, $\circ$, and atm_of_atms with $\cdot$, Var, $\lambda\sigma\,\tau\,x.\ \sigma\,x \cdot \tau$, 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 $term\ set\ set \Rightarrow ('v \Rightarrow term)\ option$. 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 ⇒ ('v ⇒ term) list
where
  renamings_apart [] = []
| renamings_apart (C # Cs) =
  let σs = renamings_apart Cs in
  (λv. v + max ({0} ∪ vars_clause_list (Cs · σs)) + 1) # σs
```

where vars_clause_list :: *term clause list* $\Rightarrow$ $'v$ *set* returns the variables contained in a list of clauses. The creation of fresh variable names relies on $'v = nat$.

Finally, the *FO_resolution_prover* locale requires that the type of atoms supports two comparison operators: a well-order $>$ and a comparison $\succ$ that is stable under substitution (i.e., $B \succ A \Longrightarrow B \cdot \sigma \succ A \cdot \sigma$). Moreover, $>$ and $\succ$ must coincide on ground atoms. We instantiate $\succ$ with the Knuth–Bendix order (KBO) [18] on terms, provided by IsaFoR [43]. KBO 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 $\succ$ to a well-order, using Hilbert choice. This makes $>$ nonexecutable, but this is acceptable since it is $>$, not $\succ$, that is used in the prover's code.

Clause Subsumption. The second hurdle concerns clause subsumption. Its mathematical definition, subsumes $C\ D \longleftrightarrow \exists\sigma.\ C \cdot \sigma \subseteq D$, involves an infinite quantification.

The problem of deciding whether such a substitution exists is NP-complete [17]. 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 ⇒ ('v ⇒ 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 nondeterminism 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 $(\lambda x.$ 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. Following Tammet [46], 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 below:

```
fun subsumes_list′ :: term literal list ⇒
  term literal list ⇒ ('v ⇒ term option) ⇒ bool
where
  subsumes_list′ [] Ks σ = True
| subsumes_list′ (L # Ls) Ks σ =
  let Ks = filter (leq_lit L) Ks in
  (∃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)
```

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 RP_x, 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.

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

export_code RP_x **in** *SML* **module_name** *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 `RPx : (term literal list * nat) list -> bool`. The input is the $\mathcal{N}$ 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 RP_x 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 $\mathsf{RP}_\mathsf{x}\ t = b$ 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 [13].

Empirical Evaluation. To measure the gap with the state of the art, we compare its performance with that of three other provers on a benchmark suite. TPTP (Thousands of Problems for Theorem Provers) [45] is the de facto standard library for benchmarking automatic provers. We extended RP_x with the trusted TPTP parser from Metis [14]. We benchmarked E 2.1, Vampire 4.2.2, Metis 2.4, and 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 clausifier. 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:

	Vampire	E	Metis	RP_x
Unsatisfiable	676	635	436	331
Satisfiable	158	135	91	22
Average time (s)	0.033	0.014	0.637	3.126

The detailed results of the evaluation are available online, together with instructions for reproducing them.[4]

As expected, RP_x is not competitive. A prover's performance comes from its calculus, its heuristics, and its indexing data structures. 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 [9].

7 Discussion and Related Work

We found Bachmair and Ganzinger's [2] chapter and its formalization [37, 38] 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.

It took us quite some time to design a suitable measure to prove the fairness of the layer 2 prover 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 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 **qed** as quickly as possible but rather to investigate how to harness a modern proof

[4] http://matryoshka.gforge.inria.fr/pubs/fun_rp_data.tar.gz

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 [31] uses superposition provers and SMT (satisfiability modulo theories) solvers to swiftly identify which lemmas are necessary to prove a goal; standard Isabelle tactics are then used to certify the proof. Isabelle's support for coinductive methods, including the **coinductive**, **codatatype**, and **corec** commands, helps us reason about infinite processes. Locales are a useful abstraction for defining the refinement layers. And Isabelle's libraries, the *Archive of Formal Proofs*, and IsaFoR certainly saved us months of labor.

The *Archive* also includes a refinement framework [24], which has been used in a separate effort to connect the imperative code of an efficient SAT solver to an abstract calculus [5]. 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 author.

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 [32] 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 [9] 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 [38]), even though they provide few details beyond layer 1.

We built the prover on our earlier formalization [37, 38] of ordered resolution. Related efforts developed using Isabelle/HOL include Peltier's [32] formalization of superposition and Schlichtkrull's [36] 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 [12] 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 Margetson's [34] verified first-order prover, based on a sequent calculus for first-order logic without full first-order terms but only variables. Kumar et al. [23] formalized the soundness of a proof assistant for higher-order logic. Jensen et al. [15] 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 [5, 26–28, 30, 41]. SAT being a decidable problem, termination has been proved for most solvers. First-order logic, on the other hand, is semidecidable, which is partly what makes our present work original.

A pragmatic approach to combine 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 [8, 25]. Researchers from the first-order theorem proving community are now advocating this approach for their systems [33]. An ad hoc version of this approach is used in Sledgehammer and HOLyHammer to reconstruct proofs found by external provers [3, 16].

8 Conclusion

Starting from an abstract description of an ordered resolution prover [37, 38], 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 for this kind of work. 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 [7, 11], first-order [20, 39, 49], and higher-order logic [42].

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, Jørgen Villadsen, and our anonymous friends at POPL 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, LIGHTest). 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 [35] with the same authors' list.

References

[1] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. 1989. Completion without Failure. In *Rewriting Techniques—resolution of Equations in Algebraic Structures*, Hassan Aït-Kaci and Maurice Nivat (Eds.). Vol. 2. Academic Press, 1–30.

[2] Leo Bachmair and Harald Ganzinger. 2001. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier and MIT Press, 19–99.

[3] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka, and Albert Steckermeier. 2016. Semi-intelligible Isar Proofs from Machine-Generated Proofs. *J. Autom. Reasoning* 56, 2 (2016), 155–200.

[4] Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. 2017. Friends with Benefits: Implementing Corecursion in Foundational Proof Assistants. In *ESOP 2017*, Hongseok Yang (Ed.). LNCS, Vol. 10201. Springer, 111–140.

[5] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. 2018. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *J. Autom. Reasoning* 61, 1–4 (2018), 333–365.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011*, K. Rustan M. Leino and Michał Moskal (Eds.). 53–64.

[7] Koen Claessen and Nicholas Smallbone. 2018. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In *IJCAR 2018 (LNCS)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.), Vol. 10900. Springer, 388–404.

[8] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient Certified RAT Verification. In *CADE-26*, Leonardo de Moura (Ed.). LNCS, Vol. 10395. Springer, 220–236.

[9] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. 2018. A Verified SAT Solver with Watched Literals using Imperative HOL. In *CPP 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 158–171.

[10] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *FLOPS 2010*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). LNCS, Vol. 6009. Springer, 103–117.

[11] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. 1997. WALDMEISTER—High-Performance Equational Deduction. *J. Autom. Reasoning* 18, 2 (1997), 265–270.

[12] Nao Hirokawa, Aart Middeldorp, Christian Sternagel, and Sarah Winkler. 2017. Infinite Runs in Abstract Completion. In *FSCD 2017*, Dale Miller (Ed.). LIPIcs, Vol. 84. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 19:1–19:16.

[13] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *ESOP 2018*, Amal Ahmed (Ed.). LNCS, Vol. 10801. Springer, 999–1026.

[14] Joe Hurd. 2003. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA) (NASA Technical Reports)*, Myla Archer, Ben Di Vito, and César Muñoz (Eds.). 56–68.

[15] Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen. 2018. Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL. *AICommun.* 31, 3 (2018), 281–299.

[16] Cezary Kaliszyk and Josef Urban. 2013. PRocH: Proof Reconstruction for HOL Light. In *CADE-24*, Maria Paola Bonacina (Ed.). LNCS, Vol. 7898. Springer, 267–273.

[17] Deepak Kapur and Paliath Narendran. 1986. NP-Completeness of the Set Unification and Matching Problems. In *CADE-8*, Jörg H. Siekmann (Ed.). LNCS, Vol. 230. Springer, 489–495.

[18] Donald E. Knuth and Peter B. Bendix. 1970. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*, John Leech (Ed.). Pergamon Press, 263–297.

[19] Laura Kovács and Andrei Voronkov. 2009. Finding Loop Invariants for Programs over Arrays using a Theorem Prover. In *SYNASC 2009*, Stephen M. Watt, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, and Daniela Zaharie (Eds.). IEEE Computer Society, 10.

[20] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *CAV 2013*, Natasha Sharygina and Helmut Veith (Eds.). LNCS, Vol. 8044. Springer, 1–35.

[21] Alexander Krauss. 2006. Partial Recursive Functions in Higher-Order Logic. In *IJCAR 2006*, Ulrich Furbach and Natarajan Shankar (Eds.). LNCS, Vol. 4130. Springer, 589–603.

[22] Alexander Krauss. 2010. Recursive Definitions of Monadic Functions. *EPTCS* 43 (2010), 1–13.

[23] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. 2016. Self-Formalisation of Higher-Order Logic: Semantics, Soundness, and a Verified Implementation. *J. Autom. Reasoning* 56, 3 (2016), 221–259.

[24] Peter Lammich. 2013. Automatic Data Refinement. In *ITP 2013*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). LNCS, Vol. 7998. Springer, 84–99.

[25] Peter Lammich. 2017. The GRAT Tool Chain—Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *SAT 2017*, Serge Gaspers and Toby Walsh (Eds.). LNCS, Vol. 10491. Springer, 457–463.

[26] Stephane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq.* Ph.D. Dissertation. Université Paris-Sud.

[27] Filip Marić. 2008. Formal Verification of Modern SAT Solvers. *Archive of Formal Proofs* (2008). Formal Proof Development. http://isa-afp.org/entries/SATSolverVerification.html.

[28] Filip Marić. 2010. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoret. Comput. Sci.* 411, 50 (2010), 4333–4356.

[29] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS, Vol. 2283. Springer.

[30] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. 2012. `versat`: A Verified Modern SAT Solver. In *VMCAI 2012*, Viktor Kuncak and Andrey Rybalchenko (Eds.). LNCS, Vol. 7148. Springer,

363–378.

[31] Lawrence C. Paulson and Jasmin Christian Blanchette. 2012. Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *IWIL-2010*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EPiC Series in Computing, Vol. 2. EasyChair, 1–11.

[32] Nicolas Peltier. 2016. A Variant of the Superposition Calculus. *Archive of Formal Proofs* (2016). Formal Proof Development. http://isa-afp.org/entries/SuperCalc.html.

[33] Giles Reger and Martin Suda. 2017. Checkable Proofs for First-Order Theorem Proving. In *ARCADE 2017*, Giles Reger and Dmitriy Traytel (Eds.). EPiC Series in Computing, Vol. 51. EasyChair, 55–63.

[34] Tom Ridge and James Margetson. 2005. A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic. In *TPHOLs 2005*, Joe Hurd and Tom Melham (Eds.). LNCS, Vol. 3603. Springer, 294–309.

[35] Anders Schlichtkrull. 2018. *Formalization of Logic in the Isabelle Proof Assistant.* Ph.D. Dissertation. Technical University of Denmark.

[36] Anders Schlichtkrull. 2018. Formalization of the Resolution Calculus for First-Order Logic. *J. Autom. Reasoning* 61, 1–4 (2018), 455–484.

[37] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. 2018. Formalization of Bachmair and Ganzinger's Ordered Resolution Prover. *Archive of Formal Proofs* (2018). Formal Proof Development. http://isa-afp.org/entries/Ordered_Resolution_Prover.html.

[38] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. 2018. Formalizing Bachmair and Ganzinger's Ordered Resolution Prover. In *IJCAR 2018*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). LNCS, Vol. 10900. Springer, 89–107.

[39] Stephan Schulz. 2013. System Description: E 1.8. In *LPAR-19*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). LNCS, Vol. 8312. Springer, 735–743.

[40] Stephan Schulz and Martin Möhrmann. 2016. Performance of Clause Selection Heuristics for Saturation-Based Theorem Proving. In *IJCAR 2016 (LNCS)*, Nicola Olivetti and Ashish Tiwari (Eds.), Vol. 9706. Springer, 330–345.

[41] Natarajan Shankar and Marc Vaucher. 2011. The Mechanical Verification of a DPLL-Based Satisfiability Solver. *Electr. Notes Theor. Comput. Sci.* 269 (2011), 3–17. LSFA 2010.

[42] Alexander Steen and Christoph Benzmüller. 2018. The Higher-Order Prover Leo-III. In *IJCAR 2018 (LNCS)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.), Vol. 10900. Springer, 108–116.

[43] Christian Sternagel and René Thiemann. 2013. Formalizing Knuth-Bendix Orders and Knuth-Bendix Completion. In *RTA 2013*, Femke van Raamsdonk (Ed.). LIPIcs, Vol. 21. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 287–302.

[44] Christian Sternagel and René Thiemann. 2018. First-Order Terms. *Archive of Formal Proofs* (2018). Formal Proof Development. http://isa-afp.org/entries/First_Order_Terms.html.

[45] Geoff Sutcliffe. 2017. The TPTP Problem Library and Associated Infrastructure: From CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning* 59, 4 (2017), 483–502.

[46] Tanel Tammet. 1998. Towards Efficient Subsumption. In *CADE-15*, Claude Kirchner and Hélène Kirchner (Eds.). LNCS, Vol. 1421. Springer, 427–441.

[47] René Thiemann and Christian Sternagel. 2009. Certification of Termination Proofs using CeTA. In *TPHOLs 2009*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). LNCS, Vol. 5674. Springer, 452–468.

[48] Andrei Voronkov. 2014. AVATAR: The Architecture for First-Order Theorem Provers. In *CAV 2014*, Armin Biere and Roderick Bloem (Eds.). LNCS, Vol. 8559. Springer, 696–710.

[49] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS Version 3.5. In *CADE-22 (LNCS)*, Renate A. Schmidt (Ed.), Vol. 5663. Springer, 140–145.

[50] Makarius Wenzel. 2012. Isabelle/jEdit—a Prover IDE within the PIDE Framework. In *CICM 2012*, Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge (Eds.). LNCS, Vol. 7362. Springer, 468–471.

[51] Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (1971).