A Verified First-Order Prover Based on Ordered Resolution

ANONYMOUS AUTHOR(S)

First-order theorem provers based on superposition, such as E, SPASS, and Vampire, play an important role in formal software verification. They are based on a sophisticated logical calculus that combines ordered resolution with equality reasoning. They also employ advanced algorithms, data structures, and heuristics. As a step towards verifying the correctness of state-of-the-art provers, we specify, using the Isabelle/HOL proof assistant, a purely functional ordered resolution prover and formally establish its soundness and 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 can extract purely functional Standard ML code.

Additional Key Words and Phrases: automatic theorem provers, proof assistants, first-order logic, refinement

1 INTRODUCTION

Formal verification of programs aims at eliminating all bugs by mechanically checking correctness with respect to a logical specification of the program’s behavior. Automatic theorem provers based on superposition, such as E [Schulz 2013b], SPASS [Weidenbach et al. 2009], and Vampire [Kovács and Voronkov 2013], are often employed as backends in verification tools. They are used to discharge verification conditions, but also to generate loop invariants [Kovács and Voronkov 2009]. Superposition is a highly successful logical calculus for first-order logic with equality, which generalizes both ordered resolution [Bachmair and Ganzinger 2001] and ordered (unfailing) completion [Bachmair et al. 1989].

Resolution does not operate on first-order formulas but instead on sets of clauses. A clause is an \(n\)-ary disjunction of literals \(L_1 \lor \cdots \lor L_n\), whose free 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., divides(2, \(n\)). The empty clause, which is false, is denoted by \(\bot\). Resolution works by refutation: Conceptually, the calculus proves a conjecture \(\forall \bar{x}. C\) from axioms \(A\) by deriving \(\bot\) from the clause set \(A \cup \{\exists \bar{x}. \neg C\}\), indicating its unsatisfiability. The transformation of formulas to clauses is usually performed by a preprocessor [Nonnengart and Weidenbach 2001]. Compared with plain resolution, ordered resolution relies on an order on the atoms to restrict the search space. Another important difference is that it uses a redundancy criterion to discard subsumed clauses at any point; for example, \(p(x) \lor q(x)\) and \(p(5)\) are subsumed by \(p(x)\).

Using formal verification, we aim to develop trustworthy programs. But why should anyone trust verification tools? In particular, modern superposition provers are highly optimized programs that rely on sophisticated calculi, with a rich metatheory, and specialized data structures. In this paper, we propose an answer by verifying, in Isabelle/HOL [Nipkow et al. 2002], a purely functional prover based on ordered resolution. The verification relies on stepwise refinement [Wirth 1971]. Four layers are connected by three refinement steps:

- Our starting point, layer 1 (Section 4), is an abstract Prolog-style nondeterministic resolution prover in its most general form, as presented by Bachmair and Ganzinger [2001] and as formalized by Schlichtkrull et al. [2018a,b]. It operates on possibly infinite sets of clauses. It establishes soundness and completeness results that are inherited by the other layers.
- Layer 2 (Section 5) is similar but operates on finite multisets of clauses and introduces a priority queue to ensure that logical inferences are performed in a fair manner, guaranteeing completeness: Given a valid conjecture, the prover will eventually find a proof.
Layer 3 (Section 6) is a deterministic program that works on finite lists, committing to a concrete 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 some auxiliary notions.

Finally, layer 4 (Section 7) 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 [Haftmann and Nipkow 2010]. The resulting prover serves first and foremost as a proof of concept: It uses an efficient calculus (layer 1) and a reasonable strategy to ensure fairness (layers 2 and 3), but it depends on naive 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, spanning much of computer science. 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, completeness is a liveness property: The program will always terminate normally with a proof if the conjecture is valid. Our executable functional prover demonstrates that, far from being academic exercises, Bachmair and Ganzinger’s [2001] framework and its formalization by Schlichtkrull et al. [2018a,b] accurately capture the metatheory of real 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 this context. This methodology has been used to verify SAT solvers [Blanchette et al. 2018; Marić 2010; Oe et al. 2012], 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 poses challenges when transferring completeness results across refinement layers.

Our contributions are as follows:

- We unveil a verified sound and complete first-order prover based on ordered resolution.
- We propose a general methodology, using modern tools, for refining an abstract Prolog-style definition of a refutational prover to an ML-style functional program, applicable to provers and other nondeterministic semidecision procedures that can be stated abstractly.
- We present a reusable library of Isabelle/HOL definitions, lemmas, and proofs that supports the methodology. These concern atoms, terms, substitutions, and derivation chains.

In addition, we offer a few “proof pearls”—smaller proving puzzles that illustrate specific techniques and that we find particularly instructive or elegant.

Our work is part of the IsaFoL (Isabelle Formalization of Logic) project, which aims at developing a library of results about logic and automated reasoning. The Isabelle source files are available in the IsaFoL repository and in the Archive of Formal Proofs. 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 the Isabelle/jEdit [Wenzel 2012] development environment, as explained in the repository’s readme file. This will ensure that logical symbols are rendered properly and will let you inspect proof states. The files were created using Isabelle version 2017, but the repositories will be updated to track Isabelle’s evolution.

1https://bitbucket.org/isafol/isafol/wiki/Home
3https://devel.isa-afp.org/entries/Ordered_Resolution_Prover.html

2 ISABELLE/HOL

Isabelle [Nipkow and Klein 2014; Nipkow et al. 2002] is a generic proof assistant that supports multiple object logics. Its most developed instantiation, Isabelle/HOL, provides a version of classical higher-order logic (HOL) [Church 1940] that supports rank-1 (top-level) polymorphism, Haskell-style type classes, and Hilbert’s choice operator. Unlike the type theories that underlie Agda [Bove et al. 2009] and Coq [Bertot and Castéran 2004], HOL has no built-in notion of computation or executability. Nonetheless, a substantial fragment of HOL corresponds closely to Standard ML or Haskell and can be exported to these languages using a code generator [Haftmann and Nipkow 2010].

Isabelle’s syntax is inspired by both ML and traditional mathematical conventions. The types are built from type variables ’a, ’b, . . . and n-ary type constructors, normally written in postfix notation (e.g., ’a list). The infix type constructor ’a ⇒ ’b is interpreted as the (total) function space from ’a to ’b. Propositions are terms of type bool, a datatype equipped with the constructors False and True. The familiar logical symbols ∀, ∃, ¬, ∧, ∨, ⇒, ⇔, and = are functions like any other, although the quantifiers and equality on functions fall outside the executable fragment.

Isabelle adheres to a tradition initiated by the LCF system [Gordon et al. 1979]: All logical inferences are derived by a small trusted kernel, and types and functions are defined rather than axiomatized to guard against inconsistencies. Isabelle/HOL provides high-level specification mechanisms inspired by typed functional programming (e.g., ML) and logic programming (e.g., Prolog). These let us define large classes of types and operations, such as inductive datatypes, recursive functions, inductive predicates, and their coinductive counterparts. For example, the codatatype and corec commands [Biendarra et al. 2017] can be used to define codatatype and productive corecursive functions in the style of Haskell, and the coinductive command can be used to introduce coinductive predicates. Internally, Isabelle synthesizes suitable low-level nonrecursive definitions and derives the user specifications via primitive inferences. This foundational approach allows the system to provide a highly expressive, trustworthy specification language.

Isabelle proofs are expressed in a language called Isar [Wenzel 2007]. It encourages a declarative, hierarchical style reminiscent of the format suggested by Lamport [1995], but with alphanumeric labels to identify intermediate proof steps. Isar also supports low-level tactics that manipulate the proof state directly, similar to those offered by Coq and other systems [Milner 1984].

Most Isabelle formalizations are structured using locales [Ballarin 2014]. A locale is a parameterized module, similar to an ML functor. The parameters may be types or terms satisfying some assumptions. For example, Isabelle/HOL provides the following basic specifications:

```
locale semigroup =
  fixes * :: ’a ⇒ ’a ⇒ ’a
  assumes (a * b) * c = a * (b * c)
locale monoid = semigroup +
  fixes 1 :: ’a
  assumes 1 * a = a and a * 1 = a
```

The semigroup locale is parameterized by a type ’a and a binary operation * on ’a, which must be commutative. The monoid locale inherits these parameters and assumptions and enriches them with a constant 1 assumed to be left- and right-neutral. Once a locale is declared, we can enter its scope at any point in a formal development. Within a locale’s scope, we can use its parameters and assumptions in definitions, lemma statements, and proofs.

To actually use a locale, we must instantiate the parameters with concrete types and terms. For example, we can instantiate monoid by taking (’a, *, 1) to be (nat, +, 0) or (nat, ×, 1), where nat is the type of natural numbers and 0, 1, +, × have their usual semantics. Before we can retrieve the definitions and lemmas from a locale, we must discharge the assumptions (e.g., 0 + a = a for all a :: nat). If a locale is parameterized by exactly one type variable, it can be introduced as a type class instead. This can be useful to offload some bureaucracy onto the type system, but it has its limitations: As in Haskell, a type class can be instantiated with a given type at most once.
3 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 the library of clausal logic developed by Blanchette et al. [2018], which is parameterized by a type ‘a of logical atoms. Literals are defined as an inductive datatype with constructors for positive and negative literals:

```
datatype 'a literal =
  Pos 'a
  | Neg 'a
```

The type of clauses is then defined as the abbreviation ‘a clause = ‘a literal multiset, where multiset is the type constructor of finite multisets. Thus, the clause \( \neg A \lor B \), where \( A \) and \( B \) are arbitrary atoms, is represented by the multiset \( \{ \text{Neg} A, \text{Pos} B \} \), and the empty clause \( \bot \) is represented by \( \{ \} \). The complement operation is defined as \( \neg \text{Neg} A = \text{Pos} A \) and \( \neg \text{Pos} A = \text{Neg} A \) for any atom \( A \).

The truth value of ground (i.e., variable-free) atoms is given by a Herbrand interpretation: a set, of type ‘a set, of all true ground atoms. The “models” predicate \( \models \) is defined as \( I \models A \iff A \in I \). This definition is lifted to literals, clauses, and sets of clauses in the usual way:

\[
\begin{align*}
    I \models \text{Pos} A & \iff A \in I \\
    I \models \text{Neg} A & \iff A \notin I \\
    I \models L & \iff \exists C \in I. I \models C
\end{align*}
\]

A set of clauses \( D \) is satisfiable if there exists an interpretation \( I \) such that \( I \models D \).

Ordered resolution crucially 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) [Thiemann and Sternagel 2009]. 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 will usually think of the atoms as being first-order terms, which can be either a variable or a symbol applied to a list of first-order terms. Another possibility would be to use applicative first-order terms [Kennaway et al. 1996], also called \( \lambda \)-free higher-order terms. A substitution is modeled as a function from variables to terms. Substitutions can be applied to first-order terms by mapping them onto the terms’ variables.

We start by defining a locale substitution_ops that declares the basic operations on substitutions: application (\( \cdot \)), identity (\( \text{id} \)), and composition (\( \circ \)):

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

Within the locale’s scope, we introduce a number of derived concepts. Ground atoms are defined as atoms that are left unchanged by substitutions:

\[
\text{is_ground} \ A \iff \forall \sigma. \ A = A \cdot \sigma
\]

Nonstrict and strict generalization are defined as

\[
\begin{align*}
  \text{generalizes} \ A \ B & \iff \exists \sigma. \ A \cdot \sigma = B \\
  \text{strictly_generalizes} \ A \ B & \iff \text{generalizes} \ A \ B \land \neg \text{generalizes} \ B \ A
\end{align*}
\]

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
\[ \text{subsumes } C \vdash D \iff \exists \sigma \cdot C \cdot \sigma \subseteq D \]
\[ \text{strictly_subsumes } C \vdash D \iff \text{subsumes } C \vdash D \land \neg \text{subsumes } D \vdash C \]

Unifiers and MGUs are characterized as follows, where \( A : \{ a \} \) set represents a unification constraint
\( A_1 = \cdots = A_k \) and \( S : \{ a \} \) set represents a set of unification constraints:
\[ \text{is_unifier } \sigma \cdot A \iff |A \cdot \sigma| \leq 1 \]
\[ \text{is_unifiers } \sigma \cdot S \iff \forall A \in S. \text{is_unifier } \sigma \cdot A \]
\[ \text{is_mgu } \sigma \cdot S \iff \text{is_unifiers } \sigma \cdot S \land (\forall \tau. \text{is_unifiers } \tau \cdot S \Rightarrow \exists \gamma. \tau = \sigma \circ \gamma) \]

The next locale, substitution, characterizes the substitution_ops operations using assumptions. 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 share no variables) and a function that, given a list of atoms, constructs an atom with these as subterms:

```plaintext
locale substitution = substitution_ops +
  fixes
  renamings_apart :: \{ a clause list \} \Rightarrow \{ s list \} \underline{ and }
  atm_of_atms :: \{ a list \} \Rightarrow \{ a \}
  assumes
  A \cdot \text{id} = A \underline{ and }
  A \cdot (\sigma \circ \tau) = (A \cdot \sigma) \cdot \tau \underline{ and }
  (\forall A. A \cdot \sigma = A \cdot \tau) \Rightarrow \sigma = \tau \underline{ and }
  \text{is_ground_cls } (C \cdot \sigma) \Rightarrow \exists \tau. \text{is_ground } \tau \land C \cdot \tau = C \cdot \sigma \underline{ and }
  \text{wfP strictly_generalizes and }
  |\text{renamings_apart } Cs| = |Cs| \underline{ and }
  \rho \in \text{renamings_apart } Cs \Rightarrow \text{is_renaming } \rho \underline{ and }
  \text{var_disjoint } (Cs \cdot \text{renamings_apart } Cs) \underline{ and }
  \text{atm_of_atms } As \cdot \sigma = \text{atm_of_atms } Bs \iff \text{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 theory files for the precise definitions of all the functions and operators. 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 needed for encoding a clause into 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 \( \sigma \) given a set of unification constraints. If a unifier exists, it returns Some \( \sigma \); otherwise, it returns None.

4 BACHMAIR AND GANZINGER’S PROVER

The formalization by Schlichtkrull et al. [2018a,b] of a nondeterministic ordered resolution prover presented by Bachmair and Ganzinger [2001] forms layer 1 of our refinement. Resolution is first defined on ground terms and proved sound and complete with respect to a propositional semantics. First-order ordered resolution is then defined and proved sound, and the ground completeness result is lifted to obtain completeness of the first-order resolution prover. The resolution inference rule is \( n \)-ary, with an optional "selection" mechanism to guide the proof search, but in this paper we disable selection and hence only need to consider the case \( n = 2 \). The binary case can be implemented efficiently and forms the basis of modern provers such as E, SPASS, and Vampire.
The ordered resolution calculus is parameterized by a total order $>$ ("larger than") on atoms. The ground version of the calculus consists of the single inference rule

$$
C \lor A \lor \cdots \lor A \quad \neg A \lor D
$$

$$
\frac{}{C \lor D}
$$

where $A$ must be larger than all the atoms in $C$ and larger than or equal to all the atoms in $D$. The side condition is not necessary for soundness, but it rules out many unnecessary inferences, thereby pruning the search space of a prover based on the calculus. Because clauses are defined as multisets, the order of the literals in a clause is immaterial; $\neg A \lor B$ and $B \lor \neg A$ are the same clause.

For first-order logic, the order on atoms $>$ is extended to an order $>$ on nonground atoms so that $B > A$ if and only if for all ground substitutions $\sigma$, we have $B \cdot \sigma > A \cdot \sigma$. The nonground version of the calculus consists of the single inference rule

$$
C \lor A_1 \lor \cdots \lor A_k \quad \neg A \lor D
$$

$$
\frac{}{(C \lor D) \cdot \sigma}
$$

where $\sigma$ is the (canonical) MGU that solves the unification problem $A_1 \equiv \cdots \equiv A_k \equiv A$, each $A_i \cdot \sigma$ is strictly $>\cdot$-maximal with respect to the atoms in $C \cdot \sigma$, and $A \cdot \sigma$ is $>\cdot$-maximal with respect to the atoms in $D \cdot \sigma$. An important detail is that to achieve completeness, the rule must be adapted slightly to rename apart the variables occurring in different premises.

Resolution works by saturation. A set of clauses $D$ is saturated if any conclusion from premises in $D$ is already in $D$. The ordered resolution calculus is refutationally complete, meaning that any unsatisfiable saturated set of clauses necessarily contains $\bot$.

Resolution provers exploit the calculus’s completeness in the following way. They 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 on the available clauses, the set reaches saturation at the limit; if the set is unsatisfiable, this means $\bot$ is eventually derived, after finitely many steps. Crucially, not only do efficient provers add clauses to their working set, they also remove clauses that are deemed redundant. This requires a refined notion of saturation. We call a set of clauses $D$ saturated upto redundancy, formally saturated _upto_ $D$, if any inference from nonredundant clauses in $D$ yields a redundant conclusion.

Bachmair and Ganzinger’s nondeterministic first-order prover, called RP, captures the “dynamic” aspects of saturation. It builds on the first-order ordered resolution rule and a redundancy criterion. The redundant clauses are those that are tautological (i.e., clauses whose literals include both $A$ and $\neg A$ for some $A$) and those that are subsumed by another clause in the working set—for example, the clauses $B$ and $A \lor B$ are both redundant if the working set already contains $B$. Furthermore, RP takes advantage of subsumption resolution, which can be expressed as an inference rule:

$$
\frac{D \lor L \quad \neg (L \lor C \lor D) \cdot \sigma}{(C \lor D) \cdot \sigma}
$$

The conclusion subsumes the second premise, which may therefore be deleted.

The RP prover is defined as an inductive predicate $\Rightarrow$ on states, where a state is a triple $S = (N', P, O)$ of new clauses $N'$, processed clauses $P$, and old clauses $O$. Initially, $N'$ is the input problem (including the negated conjecture), and $P \cup O$ is empty. Clauses can be removed if they are tautological or subsumed or if subsumption resolution is applicable. When all clauses in $N'$ have been processed (either removed entirely or moved to $P$), a clause from $P$ can be chosen for inference computation: This clause is moved to $O$, and all its conclusions with premises from the other old clauses are introduced to form the new $N'$.

In Isabelle, an inductive predicate is specified as a set of Horn-style introduction rules, as in Prolog, but with the conclusion on the right. RP is defined as follows:

---

inductive ℓ :: 'a state ⇒ 'a state ⇒ bool where
  Neg A ∈ C ∩ Pos A ∈ C ⇒ (N ∪ {C}, P, O) ~→₁ (N, P, O) 
  | D ∈ P ∪ O ∧ subsumes D C ⇒ (N ∪ {C}, P, O) ~→₂ (N, P, O) 
  | D ∈ N ∧ strictly_subsumes D C ⇒ (N, P ∪ {C}, O) ~→₃ (N, P, O) 
  | D ∈ N ∧ strictly_subsumes D C ⇒ (N, P, O ∪ {C}) ~→₄ (N, P, O) 
  | D ∈ P ∪ O ∧ reduces D C L ⇒ (N ∪ {C ∪ L}, P, O) ~→₅ (N ∪ {C}, P, O) 
  | D ∈ N ∧ reduces D C L ⇒ (N, P, O ∪ {C ∪ L}) ~→₆ (N, P, O ∪ {C}) 
  | (N ∪ {C}, P, O) ~→₇ (N, P ∪ {C}, O) 
  | (∅, P ∪ {C}, O) ~→₈ (concl_of` infers_between O C, P, O ∪ {C})

Subscripts on ~→ identify the rules. The notation f `X stands for the image of the set (or multiset) X under function f, infers_between O C calculates all the inferences whose premises are a subset of O ∪ {C} that contains C, C ∪ D denotes C and D’s multiset union, and reduces D C L is defined as

∃D’ L’ σ. D = D’ ⊔ {L’} ∧ −L = L’ ∨ σ ∧ D’ ∙ σ ⊆ C.

**Example 4.1.** There are many ways to derive ⊥ from the unsatisfiable clause set \{p(x), ¬p(a) ∨ ¬p(b)\}. The derivation on the left-hand side below relies on the two mandatory rules (rules 8 and 9).

On the right-hand side, we show a shorter derivation that exploits reduction and subsumption to avoid performing resolution inferences. In both cases, we assume a reasonable term order.

\[
\begin{align*}
(\{p(x), ¬p(a) ∨ ¬p(b)\}, \{\}, \{\}) & \to₈ (\{p(x), ¬p(a) ∨ ¬p(b)\}, \{\}, \{\}) \\
(\{p(x), ¬p(a) ∨ ¬p(b)\}, \{p(x)\}, \{\}) & \to₈ (\{¬p(a) ∨ ¬p(b)\}, \{p(x)\}, \{\}) \\
(\{\}, \{p(x), ¬p(a) ∨ ¬p(b)\}, \{\}) & \to₅ (\{¬p(b)\}, \{p(x)\}, \{\}) \\
(\{\}, \{¬p(a) ∨ ¬p(b)\}, \{p(x)\}) & \to₅ (\{⊥\}, \{p(x)\}, \{\}) \\
(\{¬p(a)\}, \{\}, \{p(x), ¬p(a) ∨ ¬p(b)\}) & \to₃ (\{⊥\}, \{\}, \{\}) \\
(\{\}, \{p(x), ¬p(a) ∨ ¬p(b)\}, \{p(x), ¬p(a)\}) & \to₈ (\{\}, \{\}, \{\}) \\
(\{\}, \{p(x), ¬p(a) ∨ ¬p(b), ¬p(a)\}, \{\}) & \to₉ (\{\}, \{\}, \{\}) \\
(\{\}, \{p(x), ¬p(a) ∨ ¬p(b), ¬p(a), ⊥\}) & \to₉ (\{\}, \{\}, \{\}) \\
\end{align*}
\]

**Example 4.2.** The next example shows that RP can diverge even on unsatisfiable clause sets:

\[
\begin{align*}
(¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)), \{\}, \{\}) & \to₉ (\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}, \{\}) \\
(\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}), \{\}) & \to₈ (\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}, \{\}) \\
(\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}), \{\}) & \to₈ (\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}, \{\}) \\
(\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}), \{\}) & \to₈ (\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}, \{\}) \\
(\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}), \{\}) & \to₈ (\{\}, \{¬p(a, a), p(x, x), ¬p(f(x), y) ∨ p(x, y)\}, \{\}) \\
\end{align*}
\]

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 employing a fair strategy for moving clauses from P to O.

Formally, a derivation is a possibly infinite sequence of states S₀ ~ S₁ ~ S₂ ~ ⋯. In Isabelle, this is expressed by the codatatype of lazy lists:

```
codatatype 'a list =
  LNil
  | LCons 'a ('a list)
```

When working with Herbrand interpretations, the canonical way of expressing the unsatisfiability weights to newer clauses, we can guarantee that all derivations are fair.

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

\[ \text{coinductive} \quad \text{chain} :: (\forall a \Rightarrow \forall b \Rightarrow \text{bool}) \Rightarrow \forall \text{llist} \Rightarrow \text{bool} \quad \text{where} \]

\[ \text{chain} R (\text{LCons} x \text{LNil}) \]

\[ \text{chain} R x s \wedge R x (\text{lhds} x s) \Rightarrow \text{chain} R (\text{LCons} x x s) \]

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 limit of a sequence \( S \) 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:

\[ \text{definition} \quad \text{Liminf} :: \forall \text{a set} \Rightarrow \forall \text{a set} \]

\[ \text{Liminf} Xs = \bigcup_{i < |Xs|} \bigcap_{j \leq i < |Xs|} \text{lnth} Xs j \]

Operators such as \( \text{Liminf} \) and \( \text{grounding_of} \) are lifted pointwise to states. For example, the limit of a sequence of states is defined as \( \text{Liminf} S = (\text{Liminf} Ns, \text{Liminf} Ps, \text{Liminf} Os) \), where \( Ns, Ps, \) and \( Os \) are the projections of the \( N, P, \) and \( O \) components of \( Ss \). For the rest of this section, we assume that \( Ss \) is a derivation.

The soundness theorem states that if \( RP \) derives \( \perp \) (represented by the multiset \( \{ \} \) ) from a set of clauses, that set must be unsatisfiable:

\[ \text{theorem} \quad \text{RP_sound}: \]

\[ \{ \} \in \text{Liminf} Ss \Rightarrow \neg \text{satisfiable} (\text{grounding_of} (\text{lhds} Ss)) \]

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

\[ \text{theorem} \quad \text{RP_model}: \]

\[ S \sim S' \Rightarrow (I \models \text{grounding_of} S' \iff I \models \text{grounding_of} S) \]

When working with Herbrand interpretations, the canonical way of expressing the unsatisfiability of a set of first-order clauses 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 in \( N \) or \( P \). Accordingly, fairness is defined as \( \text{Liminf} Ns = \text{Liminf} Ps = \{ \} \). The completeness theorem states that the limit of a fair derivation is saturated:

\[ \text{theorem} \quad \text{RP_satuated_if_fair}: \]

\[ \text{fair} Ss \Rightarrow \text{saturated_upto} (\text{Liminf} (\text{grounding_of} Ss)) \]

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

\[ \text{corollary} \quad \text{RP_complete_if_fair}: \]

\[ \text{fair} Ss \And \neg \text{satisfiable} (\text{grounding_of} (\text{lhds} Ss)) \Rightarrow \{ \} \in \text{O_of} (\text{Liminf} Ss) \]

5 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 \( P \) component of a state—as a priority queue, where lighter clauses are chosen before heavier clauses. By assigning heavier weights to newer clauses, we can guarantee that all derivations are fair.

Another necessary ingredient for completeness is that derivations must be complete; for example, the incomplete derivation consisting of the single state \( (\{ C \}, \{ \}, \{ \}) \) is not fair because \( C \) is never processed. This requirement is expressed formally as \( \text{full_chain} (\sim_w) Ss \), where the \( \text{full_chain} \) predicate is defined coinductively as

\[ \text{Liminf} Xs = \bigcup_{i < |Xs|} \bigcap_{j \leq i < |Xs|} Xs j \]

\[ \text{Operators such as} \text{Liminf} \text{and} \text{grounding_of} \text{are lifted pointwise to states. For example, the limit of a sequence of states is defined as} \text{Liminf} S = (\text{Liminf} Ns, \text{Liminf} Ps, \text{Liminf} Os), \text{where} Ns, Ps, \text{and} Os \text{are the projections of the} N, P, \text{and} O \text{components of} Ss. \text{For the rest of this section, we assume that} Ss \text{is a derivation.} \]

\[ \text{The soundness theorem states that if} RP \text{derives} \perp \text{(represented by the multiset} \{\} \text{)} \text{from a set of clauses, that set must be unsatisfiable:} \]

\[ \text{theorem} \quad \text{RP_sound}: \]

\[ \{\} \in \text{Liminf} Ss \Rightarrow \neg \text{satisfiable} (\text{grounding_of} (lhds Ss)) \]

\[ \text{A stronger, finer-grained notion of soundness relates models before and after a transition:} \]

\[ \text{theorem} \quad \text{RP_model}: \]

\[ S \sim S' \Rightarrow (I \models \text{grounding_of} S' \iff I \models \text{grounding_of} S) \]

\[ \text{When working with Herbrand interpretations, the canonical way of expressing the unsatisfiability of a set of first-order clauses is as the unsatisfiability of its grounding.} \]

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

\[ \text{theorem} \quad \text{RP_satuated_if_fair}: \]

\[ \text{fair} Ss \Rightarrow \text{saturated_upto} (\text{Liminf} (\text{grounding_of} Ss)) \]

\[ \text{In particular, if the initial problem is unsatisfiable,} \perp \text{must appear in the} O \text{component of the limit of any fair derivation:} \]

\[ \text{corollary} \quad \text{RP_complete_if_fair}: \]

\[ \text{fair} Ss \And \neg \text{satisfiable} (\text{grounding_of} (lhds Ss)) \Rightarrow \{\} \in \text{O_of} (\text{Liminf} Ss) \]
coinductive full_chain :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
    (∀y. ¬ R x y) ⇒ full_chain R (LCons x LNil)
| full_chain R xs ∧ R x (lhd xs) ⇒ full_chain R (LCons x xs)

and characterized by the equivalence

lemma full_chain_iff_chain:
    full_chain R xs ⇔ chain R xs ∧ (finite xs ⇒ ∀y. ¬ R (llast xs) y)

For the rest of this section, we fix a full chain $S$ such that $P_{of}$ (lhd $S$) = $O_{of}$ (lhd $S$) = $\{\}$. 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 $S$.

5.1 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. As a result, the $RP_w$ prover represents clauses by a pair $(C, i)$, where $i$ is the timestamp—the larger the timestamp, the younger 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:

```
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 younger ones:

```
locale weighted_FO_resolution_prover = FO_resolution_prover +
  fixes weight :: 'a wclause ⇒ nat
  assumes i < j ⇒ weight (C, i) < weight (C, j)
```

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

```
inductive ~w :: 'a wstate ⇒ 'a wstate ⇒ bool where
    Neg A ∈ C ⇒ Pos A ∈ C ⇒ (N ⊔ {(C, i)}, P, O, t) ~w1 (N, P, O, t)
| D ∈ fst '(P ⊔ O) ∧ subsumes D C ⇒ (N + {(C, i)}, P, O, t) ~w2 (N, P, O, t)
| D ∈ fst 'N ∧ C ∈ fst 'P ∧ strictly_subsumes D C ⇒
    (N, P, O, t) ~w3 (N, {(E, k) ∈ P. E ≠ C}, O, t)
| D ∈ fst 'N ∧ strictly_subsumes D C ⇒ (N, P, O ⊔ {(C, i)}, t) ~w4 (N, P, O, t)
| D ∈ fst '(P ⊔ O) ∧ reduces D C L ⇒
    (N ⊔ {(C ⊔ {L, i}), P, O, t) ~w5 (N ⊔ {(C, i)}, P, O, t)
| D ∈ fst 'N ∧ reduces D C L ∧ (∃j. (C ⊔ {L, j}) ∈ P ⇒ j ≤ i) ⇒
    (N, P ⊔ {(C ⊔ {L, i}), O, t) ~w6 (N, P ⊔ {(C, i)}, O, t)
| D ∈ fst 'N ∧ reduces D C L ⇒ (N, P, O ⊔ {(C ⊔ {L, i}), t) ~w7 (N, P ⊔ {(C, i)}, O, t)
| (N ⊔ {(C, i)}, P, O, t) ~w8 (N, P ⊔ {(C, i)}, O, t)
| (∀(D, j) ∈ P. weight (C, i) ≤ weight (D, j)) ∧
    w ~mset_set ((λD. (D, t)) 'concl_of inners_between (set_mset (fst 'O)) C) ⇒
    (I, P ⊔ {(C, i)}, O, t) ~w9 (N, {(D, j) ∈ P. D ≠ C}, O ⊔ {(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. Each $RP_w$ rule $i$ corresponds to RP rule $i$.
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 help eliminate some unfair derivations:

- The finiteness condition guarantees that the initial clause set is countable and hence that each clause in \( \mathcal{N} \) gets the opportunity to move to \( \mathcal{P} \) (and further to \( \mathcal{O} \)).
- The set-based RP allows idle transitions, such as \( (\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \sim (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O}) \) where \( C \in \mathcal{N} \cap \mathcal{P} \). The use of multisets and \( \cup \) precludes such steps in RP_w.

In the inductive definition of RP_w, the last 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, we let the prover choose 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), even though they alter the clauses by removing needless literals. This works because reduction can only happen finitely many times—a \( k \)-literal clause can be reduced at most \( k \) times. Therefore, there is no danger of divergence due to an infinite chain of reductions. Incidentally, it would also be possible to assign the current \( t \) as the reduced clause’s timestamp, but this would effectively penalize the clause, for no good reason. If anything, a reduced clause becomes more interesting, not less; after all, the most interesting clause by far is \( \bot \).

Timestamps introduce a new danger. It may be the case that a clause \( C \) is in a limit (of a sequence of states or of state components) if we project away the timestamps, but that no single timestamped clause \((C, i)\) is in 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

\[
(\_, \_ \cup \{(C,0),\}, \_) \sim \\
(\_, \_ \cup \{(C,0),(C,1),\}, \_) \sim^+ \\
(\_, \_ \cup \{(C,1),(C,2),\}, \_) \sim^+ \ldots
\]

To rule this out, the RP_w rules are formulated so that they 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, which is proved by case distinction on the rules:

**lemma** preserve_min_P:

\[
S \sim_w S' \land (C, i) \in \mathcal{P}_{of} S \land (\forall k. (C,k) \in \mathcal{P}_{of} S \Rightarrow k \geq i) \land C \in \text{fst} \ ' \mathcal{P}_{of} S' \Rightarrow \\
(C, i) \in \mathcal{P}_{of} S'
\]

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**

\[
| | :: 'a ⇒ nat\ and
size_factor :: nat\ and
timestamp_factor :: nat
\]

**assumes**

timestamp_factor > 0

**begin**

**fun** weight :: 'a wclause ⇒ nat where

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

**end**
where \(|C| = \sum_{A \in C} \neg A \in C |A|\). It is easy to prove that this definition of weight is strictly monotone and hence that this locale is a sublocale of \(\text{weighted FO resolution prover}\). This gives us a correspondingly specialized version of \(\text{RP}_w\) 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. It is mentioned in a footnote in Bachmair and Ganzinger [2001, p. 44], but they require their weight function to be monotone not only in the timestamp but also in the clause size, claiming that this is necessary to ensure fairness. Although it often makes sense to prefer small clauses to large ones, our proof reveals that clause size is irrelevant for fairness, even in the presence of reductions. This demonstrates how working out the details and making all assumptions explicit using a proof assistant can help us answer theoretical questions, generating knowledge and understanding.

**Example 5.1.** The following derivation, based on the function weight \((C, i) = |C| + i\), follows the second derivation of Example 4.1:

\[
\begin{align*}
\sim_{w8} & \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0)\}, \{\}, 1 \\
\sim_{w8} & \{\{(\neg p(a) \lor \neg p(b), 0)\}, \{(p(x), 0)\}, \{\}, 1 \\
\sim_{w9} & \{\{\neg p(a) \lor \neg p(b), 0\}, \{(p(x), 0)\}, 2 \\
\sim_{w9} & \{\{\neg p(a), 2\}, \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0)\}, 3 \\
\sim_{w8} & \{\{\neg p(a), 2\}, \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0)\}, 3 \\
\sim_{w9} & \{(\bot, 3)\}, \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0), (\neg p(a), 2)\}, 4 \\
\sim_{w8} & \{(\bot, 3)\}, \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0), (\neg p(a), 2)\}, 4 \\
\sim_{w9} & \{\{\}, \{(p(x), 0), (\neg p(a) \lor \neg p(b), 0), (\neg p(a), 2), (\bot, 3)\}, 5 \\
\end{align*}
\]

Due to the weight function, the clause \(p(x)\) must be moved from \(\mathcal{P}\) to \(\mathcal{O}\) before \(\neg p(a) \lor \neg p(b)\).

### 5.2 Refinement Proofs

To lift the soundness and completeness results about RP to \(\text{RP}_w\), we must first show that any possible behavior of \(\text{RP}_w\) on states of type \(\text{wstate}\) is a possible behavior of RP on the corresponding values of type \(\text{state}\), without timestamps. Formally:

**Lemma weighted RP imp RP:**

\[
S \sim_w S' \implies \text{state of } S \sim \text{state of } S'
\]

The proof is a straightforward induction on the introduction rules of \(\text{RP}_w\), except for one important difference: Inference computation (rule 9) converts a set to a finite multiset using \(\text{mset}\). This operation is undefined for infinite sets. Thus, we must show that from a finite set of clauses, only a finite set of inferences may be performed by \(\text{inference between}\):

**Lemma finite ord FO resolution inferences between:**

\[
\text{finite } \mathcal{D} \implies \text{finite } (\text{inference between } \mathcal{D} \mathcal{C})
\]

Our formal proof caters for \(n\)-ary resolution, but in our application we only need the binary case. A binary resolution inference takes two premises, of the form \(\text{CAA} = \mathcal{C} \lor A_1 \lor \cdots \lor A_k\) and \(\mathcal{D} = \neg A \lor D\), and produces a conclusion \(\mathcal{E} = (\mathcal{C} \lor D) \cdot \sigma\). It can be represented compactly by a tuple of the form \((\text{CAA}, \mathcal{D}, A, A, E)\), where \(A = A_1 \lor \cdots \lor A_k\). We must show that the set of such tuples returned by \(\text{inference between}\) is finite, for a logic \(\mathcal{D}\) is finite.

First, observe that the \(E\) component of a tuple is fully determined by the other four components. Hence it suffices to consider tuples of the form \((\text{CAA}, \mathcal{D}, AA, A)\). Let \(\mathcal{D}^c = \mathcal{D} \cup \{C\}\), and let \(n\) be the length of the longest clause in \(\mathcal{D}^c\). Moreover, let \(\mathcal{A} = \bigcup_{D \in \mathcal{D}^c} \text{atsms of } D\) and \(\mathcal{A}A = \{B\} \setminus \text{mset } B \subseteq \mathcal{A} \lor |B| \leq n\). Then all inferences between \(\mathcal{D}\) and \(\mathcal{C}\) belong to \(\mathcal{D}^c^2 \times AA \times \mathcal{A}\), which is a cartesian product of finite sets.
5.3 Soundness and Completeness Proofs

Using the refinement theorem, it is easy to lift the RP_model theorem (Section 4) to RP_\_w:

\[ S \sim_w S' \implies (I \models \text{grounding}_w S' \iff I \models \text{grounding}_w S) \]

Completeness is considerably more difficult. We first show that the use of timestamps ensures that all full RP_\_w derivations are fair. From this fact follows unconditional completeness.

In principle, a full derivation could be unfair by virtue of being finite and ending in a state such as N_\' or P is nonempty. However, this is impossible because a transition of rule 8 or 9 could then be taken from the last state, contradicting the hypothesis that the derivation is full. Hence, finite full derivations are necessarily fair:

**lemma** fair_if_finite:

\[ \text{finite } S_s \implies \text{fair (lmap state_of } S_s) \]

There are two ways in which an infinite derivation S_s in RP_\_w could be unfair: A clause could get stuck forever in N_\', or in P. We show that the N_\' case is impossible by defining a measure on states that decreases with respect to the lexicographic extension of > on natural numbers to pairs, which is a well-founded relation. The measure is

**abbreviation** RP_basic_measure :: 'a wstate \Rightarrow nat_\^2 where

\[ \text{RP_basic_measure } (N, P, O, t) \equiv (\text{sum } (\lambda(C, \_). |C| + 1) \cdot (N \cup P \cup O)), |N|) \]

The first component of the pair is the total size of all the clauses in the state, plus 1 for each clause to ensure that empty clauses are counted. The second component is the number of clauses in N_\'.

It is easy to see why the measure is decreasing. Rule 9, inference computation, is not applicable due to our assumption that a clause remains stuck in N_\'. Rule 8, which moves a clause from N to P, decreases the measure's second component while leaving the first component unchanged. The other rules decrease the first component since they remove clauses or literals. Formally:

**lemma** weighted_RP_basic_measure_decreasing_N:

\[ S \sim_w S' \land (C, \_ ) \in N_\text{-of } S \implies (\text{RP_basic_measure } S', \text{RP_basic_measure } S) \in \text{natLess } < \text{lex} \text{natLess} \]

What about the case where the clause is stuck in P? Lemma preserve_min_P (Section 5.1) showed that in any step, either all copies of a clause C \in P are removed or the one with minimum timestamp is preserved. It follows that if a clause C is stuck in P, its minimal timestamp can only decrease over time. Since > is well founded on natural numbers, eventually a fixed minimum i will be reached and belong to the limit:

**lemma** persistent_wclause_in_P_if_persistent_clause_in_P:

\[ C \in \text{Liminf } (\text{Imap } P_\text{-of } (\text{lmap state_of } S_s)) \implies \exists i. (C, i) \in \text{Liminf } (\text{lmap } (\text{set_mset } o P_\text{-of } ) S_s) \]

Again, we define a measure, but it must also decrease when inferences are computed and new clauses appear in 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

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

Notice that RP_filtered_measure (\lambda_. True) essentially amounts to RP_basic_measure. In the formalization, we actually use RP_filtered_measure (\lambda_. True) to avoid code duplication.
Suppose the clause $C$ that is stuck in $P$ has weight $w$ in the limit, and suppose that a clause $D$ is
moved from $P$ to $O$ by the inference computation rule. That clause’s weight must be at most $w$; otherwise, it would not have been preferred to $C$.

Infinite derivations necessarily consist of segments each consisting of finitely many applications
of other rules other than rule 9 followed by an application of rule 9: $(\sim_{w_1} \circ \sim_{w_0})^\omega$. Since each
application or 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:

\[
\text{abbreviation } \text{RP\_filtered\_rel :: } (\text{nat})^2 \seteq \text{where}
\begin{align*}
\text{RP\_filtered\_rel} & \equiv \text{natLess } \langle \text{lex} \rangle \text{ natLess } \langle \text{lex} \rangle \text{ natLess } \\
\end{align*}
\]

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

\[
\text{abbreviation } \text{RP\_combined\_measure :: } \text{nat} \Rightarrow 'a \text{ wstate } \Rightarrow \text{nat} \times (\text{nat})^3 \times \text{nat}^3 \seteq \text{where}
\begin{align*}
\text{RP\_combined\_measure } w S & \equiv \\
(w + 1 - t) \text{ of } S, \text{ RP\_filtered\_measure } (\lambda(\_ , i). i \leq w) S, \text{ RP\_basic\_measure } S
\end{align*}
\]

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

\[
\text{lemma } \text{weighted\_RP\_basic\_measure\_decreasing}\_P: \\
S \sim_w S' \land Ci \in \mathcal{P} \_ of \ S \Rightarrow \\
(\text{RP\_combined\_measure } \langle \text{weight } Ci \rangle S', \text{ RP\_combined\_measure } \langle \text{weight } Ci \rangle S) \\
\in \text{natLess } \langle \text{lex} \rangle \text{ RP\_filtered\_rel } \langle \text{lex} \rangle \text{ RP\_filtered\_rel}
\]

By combining the two lemmas $\text{weighted\_RP\_basic\_measure\_decreasing}\_N$ and $\text{weighted\_RP\_basic\_measure\_decreasing}\_P$, we can prove fairness for all derivations starting with $\mathcal{P} = \mathcal{O} = \{\}$:

\[
\text{theorem } \text{weighted\_RP\_fair}: \\
\text{fair } \text{lmap state\_of } S\_S
\]

Since all derivations in $\text{RP}_w$ are fair and its derivations are also derivations of $\text{RP}$, it is trivial to
lift $\text{RP}$’s saturation and completeness theorems, $\text{RP\_saturated\_if\_fair}$ and $\text{RP\_complete\_if\_fair}$:

\[
\text{corollary } \text{weighted\_RP\_saturated}: \\
\text{saturated\_upto } (\text{Liminf } \langle \text{lmap } \text{grounding\_of } S\_S \rangle)
\]

\[
\text{corollary } \text{weighted\_RP\_complete}: \\
\sim \text{satisfiable } \langle \text{grounding\_of } \langle \text{lhd } S\_S \rangle \rangle \Rightarrow \{} \in \mathcal{O} \_ of \ \langle \text{Liminf } \langle \text{lmap } \text{state\_of } S\_S \rangle \rangle
\]

6 ELIMINATING NONDETERMINISM

The third refinement layer defines a functional program $\text{RP}_d$ that embodies a specific rule application
strategy, thereby eliminating the nondeterminism present in $\text{RP}_w$. Clauses are now represented as
lists, and multisets of clauses as lists of lists. Although the program is deterministic, some auxiliary
functions are specified mathematically and are not directly executable; making these executable is
the objective of the fourth refinement layer (Section 7).

6.1 Definition

Our prover corresponds roughly to the following pseudocode:
function \( \text{RP}_d(N, P, O, t) \) is

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

The function should be invoked with \( N \) as the input problem, \( P = O = \{\} \), and an arbitrary
timestamp \( t \). The loop is loosely modeled after the proof procedure implemented in the E prover
[Schulz 2002, Section 3].

Instead of finite multisets, the actual \( \text{RP}_d \) definition in Isabelle uses finite lists, bringing us closer
to executable code. The \# operator abbreviates the Cons constructor, and @ is the append operator.
The list-based representations compel us to introduce the following type abbreviations:

- type_synonym 'a_clause = 'a literal list
- type_synonym 'a_dclause = 'a_clause × nat
- type_synonym 'a_dstate = 'a_dclause list × 'a_dclause list × 'a_dclause list × nat

A state is a tuple \((N, P, O, t)\) as before, but with different types.

The prover is defined inside a locale that inherits \text{weighted 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 complete definition, excluding auxiliary functions:

fun \( \text{RP}_d \_\_ \text{step} : \) 'a_dstate \( \Rightarrow \) 'a_dstate where
\( \text{RP}_d \_\_ \text{step} (N, P, O, t) = \)
  if \( \exists Ci \in P @ O . \text{fst} Ci = [] \) then
    ([], [], remdups \( P @ O \), \( t + |\text{remdups} P| \))
  else
    (case \( N \) of
      [] \( \Rightarrow \)
      (case \( P \) of
        [] \( \Rightarrow \) \( (N, P, O, t) \)
        \( |P_0 \# P' \) \( \Rightarrow \)
        let
          \( (C, i) = \text{select_min_weight_clause} P_0 P' \);
          \( N = \text{map} (\lambda D . (D, t)) (\text{remdups} (\text{resolve_rename} C C)) \)
\[ \sigma \]

where \( \text{RP} \) such as the following, which can be used to replace a terminating call to define it mathematically in this way. For example, it allows us to state and prove a characterization computation diverges. Executing the function would never actually return

Some monad. The function’s result is of the form\( \text{partial-function} \). Instead, we use \( \text{function using the} \) \( \text{fun} \) command \cite{Krauss2006}, which is restricted to well-founded recursion. Since there are no guarantees that the recursion will ever terminate, we cannot introduce the quantifier is unproblematic because it ranges over a finite set, but some of the auxiliary functions rely on infinite quantification. Notably, subsumption of \( \text{quantifier is unproblematic because it ranges over a finite set, but some of the auxiliary functions} \) \( \text{all} \) \( \text{substitutions. Nonexecutable constructs are acceptable if we know that we can replace them by equivalent executable constructs further down the refinement chain; for example, an implementation of subsumption can compute a finite set of candidates for} \) \( \text{using matching, instead of blindly enumerating all possibilities.} \)

The prover’s main program is a tail-recursive function that repeatedly calls \( \text{RP}_d \) \( \text{of the form} \) \( \text{(|} \) \( \text{into} \) \( \text{a} \) \( \text{final} \) \( \text{state, of the form} \) \( \text{RP}_d \) \( \text{and prove a characterization such as the following, which can be used to replace a terminating call} \) \( \text{RP}_d \) \( \text{by a finite iteration} \) \( \text{RP}_d \) \( \text{monad. The function’s result is of the form} \) \( \text{fun} \) \( \text{command} \) \cite{Krauss2006}, which is restricted to well-founded recursion. Instead, we use \( \text{partial-function} \) \cite{Krauss2010}, 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 \( \text{RP}_d \) \( \text{by a finite iteration} \) \( \text{RP}_d \) \( \text{partial-function} \) \( \text{option} \) \( \text{RP}_d :: \ 'a \ dstate \Rightarrow \ 'a \ lclause \ list \ \text{option} \ where \)

\[
\text{RP}_d \ S = \text{if is_final} \ S \text{then Some (map fst (C_of} \ S) \text{) else RP}_d \ (\text{RP}_d \ \text{step} \ S)
\]

Since there are no guarantees that the recursion will ever terminate, we cannot introduce the function using the \( \text{fun} \) command \cite{Krauss2006}, which is restricted to well-founded recursion. Instead, we use \( \text{partial-function} \) \cite{Krauss2010}, 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 \( \text{RP}_d \) \( \text{by a finite iteration} \) \( \text{RP}_d \) \( \text{partial-function} \) \( \text{option} \) \( \text{RP}_d :: \ 'a \ dstate \Rightarrow \ 'a \ lclause \ list \ \text{option} \ where \)

\[
\text{RP}_d \ S = \text{if is_final} \ S \text{then Some (map fst (C_of} \ S) \text{) else RP}_d \ (\text{RP}_d \ \text{step} \ S)
\]

Since there are no guarantees that the recursion will ever terminate, we cannot introduce the function using the \( \text{fun} \) command \cite{Krauss2006}, which is restricted to well-founded recursion. Instead, we use \( \text{partial-function} \) \cite{Krauss2010}, 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 \( \text{RP}_d \) \( \text{by a finite iteration} \) \( \text{RP}_d \) \( \text{partial-function} \) \( \text{option} \) \( \text{RP}_d :: \ 'a \ dstate \Rightarrow \ 'a \ lclause \ list \ \text{option} \ where \)

\[
\text{RP}_d \ S = \text{if is_final} \ S \text{then Some (map fst (C_of} \ S) \text{) else RP}_d \ (\text{RP}_d \ \text{step} \ S)
\]
6.2 Refinement Proofs

Using refinement, we connect the \(R_{\text{d-step}}\) function to the \(R_{\text{p}}\) predicate. \(R_{\text{d-step}}\) has a coarser granularity than \(R_{\text{p}}\): A single invocation on a nonfinal state \(S\) can amount to a chain of \(R_{\text{p}}\) transitions. This is captured by the following weak-refinement property:

**Lemma** nonfinal_deterministic \(R_{\text{p}}\) step:

\[
\neg \text{is_final } S \implies \text{wstate of } S \xrightarrow{\text{w}} \text{wstate of } (R_{\text{d-step}} S)
\]

where wstate of converts \(R_{\text{d}}\) states to \(R_{\text{p}}\) states. The entire proof, including key lemmas, is about 1300 lines long. It follows the case distinctions present in the definition of \(R_{\text{d-step}}\):

**Case** \(\exists Ci \in P \cap O. \text{fst } Ci = []\):

By induction on \(|\text{remdups } P|\), there must exist a derivation of the form

\[
\xrightarrow{\text{w}} \text{wstate of } (N, P, O, t) \xrightarrow{\text{w}} \text{wstate of } (N', P', (C, i) \not\in O, t + 1)
\]

for \(P' = \text{filter } (\lambda(D, j). \text{mset } D \not= \text{mset } C) P\) and suitable \(N'\) and \((C, i) \in P\). The last step is justified by the induction hypothesis.

**Case** \(N = P = []\):

Contradiction with the assumption that \((N, P, O, t)\) is a nonfinal state.

**Case** \(N = []\):

It suffices to show that the transition

\[
\text{wstate of } ([], P, O, t) \xrightarrow{\text{w}} \text{wstate of } (N', P', (C, i) \not\in O, t + 1)
\]

is possible, where \((C, i) \in P\) is a minimal-weight clause and

\[
N' = \text{map } (\lambda(D, (D, t)) \text{remdups } \text{resolve rename } C C @ \text{concat } (\text{map } \text{resolve rename}_\text{either way } C \circ \text{fst } O))
\]

\[
P' = \text{filter } (\lambda(D, (D, j). \text{mset } D \not= \text{mset } C) P)
\]

The main proof obligation is that \(N'\), converted to multisets, equals the multiset \(\text{mset } (\lambda(D, (D, t)) \text{conc }\text{of }\text{infers between } (\text{set } \text{mset } (\text{fst } O) C)\text{specified in rule }\text{w}\). The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

**Otherwise**:

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

\[
\xrightarrow{\text{w}} \text{wstate of } ((C, i) \not\in N', P, O, t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } (([], i) \not\in N', P, O, t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } (([], i) \not\in N', [], O, t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } (([], i) \not\in N', [], [], t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } (N', [([], i)], [], t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } ([], [([], i)], [], t)
\]

\[
\xrightarrow{\text{w}} \text{wstate of } ([], [[]], i), t)
\]
Otherwise, if is_tautology \( C' \lor \) subsume (map fst (\( P @ O \))) \( C' \), then

\[
\text{wstate }_\sim ((C, i) \# N', P, O, t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', P, O, t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', P', O, t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', \text{back_to}_P @ P', O', t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', \text{back_to}_P @ P', O'', t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', P'', O'', t)
\]

\[
\text{wstate }_\sim ((N', (C', i) \# P''), O'', t)
\]

Otherwise:

\[
\text{wstate }_\sim ((C, i) \# N', P, O, t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', P', O, t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', \text{back_to}_P @ P', O', t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', \text{back_to}_P @ P', O'', t)
\]

\[
\text{wstate }_\sim ((C', i) \# N', P'', O'', t)
\]

\[
\text{wstate }_\sim ((N', (C', i) \# P''), O'', t)
\]

for suitable clause lists \( P' \), \( \text{back_to}_P \), \( O' \), \( O'' \), and \( P'' \).

The above refinement theorem, about computations from nonfinal states, is complemented by the following trivial result about final states:

**Lemma final_deterministic_RP_step:**

\[
\text{is_final } S \Rightarrow \text{RP}_d \text{ step } S = S
\]

### 6.3 Soundness and Completeness Proofs

Let \( \mathcal{S}_0 = (\mathcal{N}_0, [], [I], 0) \). For \( \text{RP}_d \), soundness means that whenever \( \text{RP}_d \mathcal{S}_0 \) 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 \( \perp \), which provides a simple syntactic check for satisfiability. Completeness means that divergence is possible only if \( \mathcal{N}_0 \) is satisfiable. Note that for satisfiable clause sets \( \mathcal{N}_0 \), both termination and divergence are possible.

To lift soundness and completeness results from \( \text{RP}_w \) to \( \text{RP}_d \), we first define \( \mathcal{S}_s \) as a full chain of nontrivial \( \text{RP}_d \) steps starting from \( \mathcal{S}_0 \). Formally, we let \( \mathcal{S}_s = \text{derivation_from } \mathcal{S}_0 \), with

**primecorec** derivation_from :: \( 'a \text{ dstate} \Rightarrow 'a \text{ dstate list where}

**derivation_from** \( S = \text{LCons } S \) (if \( \text{is_final } S \) then \( \text{LNil else derivation_from } \) (\( \text{RP}_d \text{ step } S \))

Based on \( \mathcal{S}_s \), we define \( w\mathcal{S}_s = \text{map wstate }_\sim \mathcal{S}_s \) and note that \( w\mathcal{S}_s \) is a full chain of \( \sim^+ \) transitions. Using a lemma that will be proved in Section 6.4, we obtain a full chain \( ssw\mathcal{S}_s \) of “small step” \( \sim_w \) transitions. This chain satisfies the conditions postulated on \( \mathcal{S}_s \) in Section 6.3, allowing us to lift the results presented there.

The soundness results are proved in a nameless locale, or context, that assumes termination:

**context**

**fixes** \( R :: 'a \text{ lclause list} \)

**assumes** \( \text{RP}_d \mathcal{S}_0 = \text{Some } R \)

The definition of \( \text{RP}_d \), using **partial_function**, gives us an induction rule restricted to the case where \( \text{RP}_d \) terminates (i.e., returns a \( \text{Some} \) 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**:

\[
I \models \text{grounding_of } \mathcal{N}_0 \iff I \models \text{grounding_of } R
\]
theorem deterministic_RP_saturated:
  saturated_upo (grounding_of R)

Admittedly, the terminology is somewhat confusing. For RP and RP\(_w\), it is natural—indeed, conform
to the literature—to classify saturation as a completeness property. However, for finite derivations,
such as those considered here, saturations amounts to a soundness property.

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 \(\mathcal{N}_0\)) \iff \{\} \in R

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

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 \{\\} \in \mathcal{O}_\text{of} \text{sws}s. It is easy to show that \text{sws}s’s limit is a subset of \text{ws}s’s limit;
hence \{\\} \in \mathcal{O}_\text{of} \text{ws}s. This implies the existence of a natural number \(k\) such that
\{\} \in \mathcal{O}_\text{of} (\text{lnth ws} k). Hence \{\} \in \mathcal{O}_\text{of} (\text{RP}_d \text{step} k \mathcal{S}_0). However, by an induction
on \(k\), we can show that \text{RP}_d must terminate after at most \(k\) iterations, contradicting
the assumption that \text{RP}_d diverges.

6.4 A Coinductive Puzzle

A single “big” step of the deterministic prover \(\text{RP}_d\) may consist of multiple “small” steps of the
weighted prover \(\text{RP}_w\). To transfer the results from \(\text{RP}_w\) to \(\text{RP}_d\), we must expand \(\text{RP}_d\)’s big steps. The
core of the expansion is an abstract property of chains and a relation’s 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 would follow by straightforward induction. But the completeness
proof must also consider infinite chains. To prove the property on infinite chains requires us to use
coinduction and corecursion up-to techniques.

The above property is formalized as follows:

lemma chain_tranclp_imp_exists_chain:
  chain \(R^+\) \(xs\) \iff \exists ys. chain \(R\) \(ys\) \land \(xs \subseteq ys\) \land \text{lhd xs} = \text{lhd ys} \land \text{llast xs} = \text{llast ys}

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

coinductive \(\subseteq::\) ‘a list \(\Rightarrow\) ‘a list \(\Rightarrow\) bool where
  lfinite \(xs\) \\iff LNil \subseteq xs
  \mid \(xs\) \subseteq \(ys\) \Rightarrow LCons \(x\) \(xs\) \subseteq zs ++ LCons \(x\) \(ys\)

fun ++ :: ‘a list \(\Rightarrow\) ‘a list \(\Rightarrow\) ‘a list where
  [] ++ \(xs\) = \(xs\)
  \mid (zs \# zs) ++ \(xs\) = LCons \(z\) (zs ++ \(xs\))

The definition of \(\subseteq\) ensures that infinite lazy lists only embed other infinite lazy lists, but not the
finite ones. Formally: \(xs \subseteq ys\) \iff (lfinite \(xs\) \iff lfinite \(ys\)). The unguarded calls to llast may seem

worrying, but it is conveniently defined to always return the same unspecified element for infinite lists (i.e., \( \neg \) finite \( xs \land \neg \) infinite \( ys \) \( \implies \) \( \text{llast} \) \( xs = \text{llast} \) \( ys \)).

To prove \( \text{chain_tranclp_imp_exists_chain} \), we instantiate the existential quantifier by the following coreursively defined witness:

\[
\text{corec} \, \text{wit} :: (\exists \, a \implies \langle a \to \text{bool} \rangle) \implies \langle a \to \text{llist} \to \text{llist} \rangle \text{ where }
\]
\[
\begin{align*}
\text{wit} \, R \, xs &= (\text{case} \, xs \ \text{of} \\
LCons \, x \, (LCons \, y \, ys) &= LCons \, x \, (\text{pick} \, R \, x \, y \; \# \text{wit} \, R \, xs) \\
\) \end{align*}
\]

Here \( \text{pick} \, R \, x \, y \) returns an arbitrary finite list of \( R \)-related intermediate states connecting the \( R^+ \)-related states \( x \) and \( y \). Formally, \( \text{pick} \, R \, x \, y = \text{SOME} \, zs \) chain \( R \) (\( \text{llist}_\text{of} \) \( (x \# zs @ [y]) \)), where \( \text{llist}_\text{of} \) converts finite lists into lazy list and \( \text{SOME} \) is Hilbert’s choice operator. Thus, \( \text{pick} \) satisfies the characteristic property \( R^+ \, x \, y \implies \text{chain} \, R \) (\( \text{llist}_\text{of} \) \( (x \# \text{pick} \, R \, x \, y \# [y]) \)). The use of Hilbert choice makes \( \text{pick} \) and \( \text{wit} \), nonexecutable; this is unproblematic since these constants are used only in proofs and not in the actual prover’s code.

The definition of \( \text{wit} \) is not primitively coreursive. Although there is a guarding \( LCons \) constructor, the coreursive call occurs under \( \# \), which makes the productivity of this function subtle. This syntactic structure of the definition is called \textit{corecursive up to \#}. What ensures \( \text{wit} \)’s productivity is the fact that \( \# \) does not removes any \( LCons \) constructors from its second arguments. A slightly weaker requirement, called \textit{friendliness} [Blanchette et al. 2015], is supported by Isabelle’s \text{corec} command [Blanchette et al. 2017]. Hence, \# must be registered as a “friend,” which involves an one-line proof, for the above definition to be accepted by Isabelle.

The four conjuncts in \( \text{chain_tranclp_imp_exists_chain} \) are then discharged separately under the common assumption chain \( R^+ \, xs \). In increasing difficulty: \( \text{lhds} \) \( \) \( \text{wit} \, R \, xs = \text{lhds} \, xs \) follows by simple rewriting. Next, \( \text{last} \) \( \) \( \text{wit} \, R \, xs = \text{llast} \, xs \) requires an induction in the case of finite chains \( xs \). For any infinite chain \( zs \) of type \( \langle a \to \text{llist} \rangle \), \( \text{llast} \, zs \) is defined a fixed but not further specified value of type \( \langle a \rangle \). The properties \( xs \sqsubseteq \text{wit} \, R \, xs \) and chain \( R \) \( \) \( \text{wit} \, R \, xs \) require a coinduction on \( \sqsubseteq \) and chain, respectively. However, in keeping with the definitional principle of corecursion up to \( \# \), plain coinduction on \( \sqsubseteq \) and chain does not suffice and we must use coinduction up to \( \# \) on \( \sqsubseteq \) and chain. We contrast the coinduction (left) and coinduction up to \( \# \) (right) rules for chain:

\[
\begin{align*}
(\exists \, z. \, xs = LCons \, z \, \text{Nil}) \lor \\
\forall xs. \, X \, xs \implies (\exists \, z. \, xs = LCons \, z \, zs \land \ X \, zs \land R \, z \, (\text{lhds} \, zs))
\end{align*}
\]

\[
\begin{align*}
(\exists \, z. \, xs = LCons \, z \, \text{Nil}) \lor \\
\forall xs. \, X \, xs \implies (\exists \, z. \, zs \, ys \, xs = LCons \, z \, (ys \# zs) \land \ X \, zs \land \text{chain} \, R \, (z \# ys @ [\text{lhds} \, zs]))
\end{align*}
\]

The property \( \text{chain_tranclp_imp_exists_chain} \) easily extends to full chains (because the last element in the case of finite chains remains unchanged), as required in Section 6.3.

### 7 Obtaining Executable Code

Our deterministic prover \( \text{RP}_d \) is already quite close to being an executable program. There are two main ingredients missing: a concrete representation of terms, over which we have abstracted so far, and an executable algorithm for clause subsumption.

#### 7.1 First-Order Terms

First-order terms are a core data structure in various fields of computer science, be it logic, rewriting, (tree) automata theory, or programming languages (as types of the simply typed \( \lambda \)-calculus). It should be no surprise that various formalizations of terms exist. We instantiate our abstract notion of atoms using a particularly comprehensive formalization of terms developed as part of the IsaFoR
library [Thiemann and Sternagel 2009]. This rewriting-independent part of IsaFoR has recently migrated to the Archive of Formal Proofs [Sternagel and Thiemann 2018].

IsaFoR terms are defined as the following datatype:

\[
\text{datatype} \quad (\langle f \rangle, \langle v \rangle) \text{ term} = \\
\quad \text{Var} \ \langle v \rangle \\
\quad \text{Fun} \ \langle f \rangle ((\langle f \rangle, \langle v \rangle) \text{ term list})
\]

To simplify notation, in this paper we fix \( f = \langle v \rangle = \text{nat} \) and work with the monomorphic type \( (\langle f \rangle, \langle v \rangle) \text{ term} \), which we abbreviate by \text{term}. In the formalization, polymorphic types are used whenever possible. IsaFoR also define the standard monadic term substitution \( \cdot :: \text{term} \Rightarrow (\langle v \rangle \Rightarrow \text{term}) \Rightarrow \text{term} \) and a function \text{unify} :: (\text{term} \times \text{term}) \text{ list} \Rightarrow \text{lsubst} \Rightarrow \text{lsubst}, where \text{lsubst} = (\langle v \times \text{term} \rangle) \text{ list} \) is the list-based representation of a finite substitution. The function \text{unify} computes the MGU for a list of unification constraints that is compatible with a given substitution. IsaFoR includes a wealth of theorems about the defined functions, including the correctness of the well-foundedness of strict term generalization \( > :: \text{term} \Rightarrow \text{term} \Rightarrow \text{bool} \) defined by \( t > s \iff (\exists \sigma. s \cdot \sigma = t) \land (\exists \rho. t \cdot \rho = s) \).

This infrastructure allows us to conveniently instantiate our locales \text{substitution_ops}, \text{substitution}, and \text{mgu}. We instantiate the type \( \langle a \rangle \) of atoms with \text{term} and the type \( \langle s \rangle \) of substitutions with \( \langle v \rangle \Rightarrow \text{term} \) and the constants \( \cdot, \text{id}, \circ, \) and \text{atm_of_atms} with \( \cdot, \text{Var}, \lambda \sigma \tau x, \sigma x \cdot \tau, \) and \text{Fun} \ 0, respectively. (The function symbol name 0 is arbitrary.) For the computation of the MGU, there is a slight type mismatch: IsaFoR offers a list-based unifier, whereas our locale requires the type \text{term set set} \Rightarrow (\langle v \Rightarrow \text{term} \rangle) \text{ option}. It is easy to translate a finite set of finite sets of terms (where the inner sets of terms are the ones to be unified) into a finite list of pairs of constraints. To be executable, the translation requires us to sort the terms contained in a set with respect to some arbitrary (but executable) linear order on terms.

Only the function \text{renamings_apart} was not present in IsaFoR. We supply this definition:

\[
\text{fun} \ \text{renamings_apart} :: \text{term clause list} \Rightarrow (\langle v \Rightarrow \text{term} \rangle) \text{ list where}
\]

\[
\begin{align*}
\text{renamings_apart} \ &= \ [] \\
\text{renamings_apart} \ (C \# Cs) \ &= \\
\quad \text{let} \\
\quad \sigma s \ &= \ \text{renamings_apart} \ Cs \\
\quad \sigma \ &= \ \lambda \sigma v. \ v + \max \ ((\{0\} \cup \text{vars_clause_list} \ C s) + 1)
\end{align*}
\]

where \text{vars_clause_list} :: \text{term clause list} \Rightarrow \langle v \rangle \text{ set} returns the variables contained in a list of clauses. The creation of fresh variable names relies on \( \langle v \rangle = \text{nat} \).

Finally, the \text{FO_resolution_prover} locale further requires that the type of atoms supports two comparison operators: a well-order > and a comparison > that is stable under substitution (i.e., \( B > A \Rightarrow B \cdot \sigma > A \cdot \sigma \)). Moreover, > and > must coincide on ground atoms. Our approach is to instantiate > with the Knuth–Bendix order (KBO) [Knuth and Bendix 1970] on terms, which is formalized in IsaFoR [Sternagel and Thiemann 2013]. 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 > to a well-order, using Hilbert choice. This makes > nonexecutable, which is unproblematic given that > is used only in proofs and not in the actual prover’s code (which relies on >).

Working with different orders poses a slight technical challenge in Isabelle. Orders are organized as type classes, which are comfortable to work with as they hide the order assumption. However, a type class can be instantiated with a concrete order at most once—in our case by >. This instantiation propagates to subsequent definitions, such as sorting or computing the minimum. To use a different
order for sorting, one has to resort to lower-level definitions that are explicitly parameterized by
the comparison operation. This is inconvenient when defining programs and even more so when
reasoning about them.

### 7.2 Clause Subsumption

The second hurdle is clause subsumption. Its mathematical definition, \( \text{subsumes} \ C \ D \iff \exists \sigma. \ C \cdot \sigma \subseteq D \), involves a nonexecutable quantification over substitutions. The problem of determining whether
such a substitution exists is \( \text{NP} \)-complete [Kapur and Narendran 1986]. We start with the following
naive code. In contrast to the mathematical definition, which operates on multisets of literals, our
function operates on lists:

```ml
fun subsumes_list :: term literal list ⇒ term literal list ⇒ osubst ⇒ 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 type declaration, \( \text{osubst} \) abbreviates \( \nu \Rightarrow \text{term option} \). The function recurses on its first
argument. In the recursive case, we must consider all possible matching literals for \( L \) from \( Ks \)
compatible with the substitution \( \sigma \). The bounded existential quantification that expresses this
 nondeterminism can be executed by iterating over the finite list \( Ks \). The functions \( \text{is_pos} \) and
\( \text{atm_of} \) are the standard discriminator and selector for literals. The function \( \text{match_term_list} \) 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, \( \text{match_term_list} \) returns
None. This substitution-passing style is typical of purely functional implementations of unification
and matching procedures and is inherited by our \( \text{subsumes_list} \).

It is easy to prove that the above executable function correctly implements clause subsumption:
\( \text{subsumes} \ (\text{mset} \ Ls) \ (\text{mset} \ Ks) = \text{subsumes_list} \ Ls \ Ks \ (λx. \text{None}) \), where \( \text{mset} \) coverts 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. This has lead to the empirical studies
of various heuristics to improve over the naive exhaustive search [Schulz 2013a; Tammet 1998].
Following Tammet [1998], we implement a simple but useful heuristic that tries to avoid needless
calls to \( \text{match_term_list} \). The heuristic replaces calls to \( \text{match_term_list} \), which are linear in the
input term sizes, with a simpler 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. Accordingly, we sort our (list-represented) clauses according to a literal quasi-order
(i.e., a transitive and reflexive relation) \( \text{leq_lit} \) such that

\[
\text{is_pos} \ L = \text{is_pos} \ K \land \text{match_term_list} [(\text{atm_of} \ L, \text{atm_of} \ K)] \ σ = \text{Some} \ ρ \Rightarrow \text{leq_lit} \ L \ K
\]

Any quasi-order with this property would refine \( \text{subsumes_list} \), by removing too small literals
(according to \( \text{leq_lit} \)) in the line highlighted in gray:

```ml
fun subsumes_list_filter :: term literal list ⇒ term literal list ⇒ osubst ⇒ bool where
subsumes_list_filter [] Ks σ = True
| subsumes_list_filter (L # Ls) Ks σ =
  let Ks = filter (leq_lit L) Ks in
```

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 $\Phi_{22}$ within 300 s, and 4 provers solve $\Phi_{20}$ within 300 s. Our verified prover solves $\Phi_{20}$ in 100 s seconds and $\Phi_{22}$ in 200 s. Although our prover cannot yet challenge state-of-the-art provers in general, its performance is respectable and could be improved further using refinement.
8 DISCUSSION AND RELATED WORK

We found Bachmair and Ganzinger’s [2001] chapter and its formalization by Schlichtkrull et al. [2018a,b] a suitable starting point for a verified prover. Nonetheless, we faced several difficulties, notably how to best divide the work into refinement steps. We developed layers 2, 3, and 4 more or less in parallel, with each of the authors working on his 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 one at a time, instead of monolithically. Another strength of refinement is that it allows us to prove results at a high level of abstraction; for example, the fairness of layer 2 is inherited by layers 3 and 4 and could be inherited by further layers. The main weakness of 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 $RP_w$. Our solution amounts to fast-forwarding to a state carrying a suitably high timestamp and filtering out all overly heavy clauses. Initially, our proof consisted of two steps—fast-forwarding and filtering—each with its own measure. This proof gave us the insurance that $RP_w$ was fair, but we found that combining the measures is both more succinct and more intelligible.

The main goal of our formalization effort was not to obtain a “QED” as quickly as possible but 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, discharge proof obligations. The Sledgehammer tool [Paulson and Blanchette 2012] 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 discharge the goal in a trustworthy way. 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 the theories included in the Isabelle distribution, the Archive of Formal Proofs, and the third-party IsaFoR libraries certainly saved us months of work.

The Archive also includes a refinement framework [Lammich 2013], which was notably used to connect the imperative code of an efficient SAT solver to an abstract calculus [Blanchette et al. 2018]. The framework is particularly helpful 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 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 use the framework. We conjecture that it could be useful to refine the prover further to obtain imperative code.

Thanks to the verification, we can trust to a very high extent that our ordered resolution prover is sound and complete. To make the prover’s performance competitive with E, SPASS, and Vampire, we would need to extend the current work along two axes. First, we should use superposition, together with its extensive simplification machinery, as the base calculus. A good starting point would be to apply our methodology to Peltier’s [2016] formalization of (a generalization of) superposition. Given that most 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 PhD thesis. Second, the chain of refinement should be continued to cover optimized algorithms and data structures. These could be specified by refining layer 4 further, along the line of Fleury et al.’s [2018] refinement of an imperative SAT solver.

In computer science, metatheories and implementations are often left unconnected. A metatheory may inspire an implementation, or vice versa, but the connection is rarely made explicit.
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 [2001] (with the exceptions noted by Schlichtkrull et al. [2018b]) accurately describe the abstract principles of an executable functional prover, even though they provide few details on how to move past layer 1.

We built our verified prover on Schlichtkrull et al.’s [2018a,b] formalization of ordered resolution. Related efforts, developed using Isabelle/HOL, including Schlichtkrull’s [2018] formalization of unordered resolution and Peltier’s [2016] formalization of a variant of the resolution calculus. However, these developments only cover logical calculi; had we started with any of them, the first step would have been to define the equivalent of layer 1 and prove basic properties about it.

Another related effort is Hirokawa et al.’s [2017] formalization of ordered completion, which (like ordered resolution) is essentially a special case of superposition.

Formalizing a theorem proving tool using a theorem proving tool is a thrilling prospect for many researchers. An early result is Ridge and Margetson’s [2005] verified first-order prover, based on a sequent calculus for first-order logic without full first-order terms but only variables. Jensen et al. [2017] verified the soundness of a kernel for an interactive theorem prover for first-order logic that includes a tableau prover. Kumar et al. [2016] formalized the soundness of an interactive theorem prover for higher-order logic. There are several verified SAT solvers [Blanchette et al. 2018; Lescuyer 2011; Marić 2008, 2010; Oe et al. 2012; Shankar and Vaucher 2011]. Among these, two implement efficient data structures such as the two watched literals [Blanchette et al. 2018; Oe et al. 2012]. SAT being a decidable problem, all the above solvers except for one [Oe et al. 2012] have been proved to terminate. First-order logic, on the other hand, is semidecidable, and this is partly what makes our present work original. Lifting, via refinement, an abstract completeness result expressed in terms of the limit of a possibly infinite derivation to a possibly nonterminating functional program is something we have not found anywhere in the literature.

A pragmatic approach to combine the efficiency of unverified code with the trustworthiness of verified code consists in checking certificates produced by reasoning tools—e.g., proofs produced by SAT solvers [Cruz-Filipe et al. 2017; Lammich 2017]. Researchers from the first-order theorem proving community [Reger and Suda 2017] are now advocating this approach for their systems as well. An ad hoc version of this approach is used in Sledgehammer and similar tools to reconstruct proofs found by first-order provers [Blanchette et al. 2016; Kaliszyk and Urban 2013].

9 CONCLUSION

Starting from Schlichtkrull et al.’s [2018b] abstract specification of an ordered resolution prover, we verified, through a chain of refinement steps, an 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.

The stepwise refinement methodology is a keystone of our approach, and we found it entirely adequate for this kind of work. Each refinement step cleanly isolates concerns, yielding intelligible proof obligations. Refinement also helped us identify a needless assumption in Bachmair and Ganzinger [2001], and generally 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 saturation-based provers, using Isabelle/HOL or other systems, will be substantially easier.

REFERENCES

A Verified First-Order Prover Based on Ordered Resolution
