A Verified Automatic Prover Based on Ordered Resolution

ANDERS SCHLICHTKRULL, Technical University of Denmark, Denmark
JASMIN CHRISTIAN BLANCHETTE, Vrije Universiteit Amsterdam, The Netherlands and Max-Planck-Institut für Informatik, Germany
DMITRIY TRAYTEL, ETH Zürich, Switzerland

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 sophisticated logical calculi that combine ordered resolution and 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 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.

CCS Concepts: • **Theory of computation** → **Logic and verification**; *Automated reasoning*; • **Software and its engineering** → *Completeness*;

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

ACM Reference Format:
Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. 2018. A Verified Automatic Prover Based on Ordered Resolution. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 27 pages.

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 $L_1 \vee \cdots \vee L_n$ 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].

Authors' addresses: Anders Schlichtkrull, DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark, andschl@dtu.dk; Jasmin Christian Blanchette, Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, Amsterdam, 1081 HV, The Netherlands, j.c.blanchette@vu.nl, Research Group 1, Max-Planck-Institut für Informatik, Saarland Informatics Campus E1 4, Saarbrücken, 66123, Germany, jasmin.blanchette@mpi-inf.mpg.de; Dmitriy Traytel, Institute of Information Security, Department of Computer Science, ETH Zürich, Universitätstrasse 6, Zürich, 8092, Switzerland, traytel@inf.ethz.ch.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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, $\mathrm{p}(x) \vee \mathrm{q}(x)$ and $\mathrm{p}(5)$ are subsumed by $\mathrm{p}(x)$.

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 a highly 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. Its soundness and refutational completeness are inherited by the other layers.
- Layer 2 (Section 5) 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, refutational 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 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 this context. This methodology has been used to verify SAT solvers [Blanchette et al. 2018; Marić 2010], 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 instructive or elegant.

Our 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 source 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 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.

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 $'a \Rightarrow 'b$ 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 $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$, and $=$ are normal functions, 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

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

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 $* :: 'a \Rightarrow 'a \Rightarrow 'a$
  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 $(nat, \times, 1)$, where *nat* is the type of natural numbers and $0, 1, +, \times$ 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 $\neg A \vee B$, where A and B are arbitrary atoms, is represented by the multiset $\{\mathsf{Neg}\ A, \mathsf{Pos}\ B\}$, and the empty clause $\bot$ is represented by $\{\}$. The complement operation is defined as $-\mathsf{Neg}\ A = \mathsf{Pos}\ A$ and $-\mathsf{Pos}\ A = \mathsf{Neg}\ A$ 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 $I \models A \mathrel{\mathop{\Longleftrightarrow}} A \in I$. This definition is lifted to literals, clauses, and sets of clauses in the usual way:

$$I \models \mathsf{Pos}\ A \mathrel{\mathop{\Longleftrightarrow}} A \in I \qquad I \models C \mathrel{\mathop{\Longleftrightarrow}} \exists L \in C.\ I \models L$$
$$I \models \mathsf{Neg}\ A \mathrel{\mathop{\Longleftrightarrow}} A \notin I \qquad I \models \mathcal{D} \mathrel{\mathop{\Longleftrightarrow}} \forall C \in \mathcal{D}.\ I \models C$$

A set of clauses $\mathcal{D}$ is *satisfiable* if there exists an interpretation I such that $I \models \mathcal{D}$.

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, also called λ-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 (id), and composition ($\circ$):

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

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

$$\text{is\_ground } A \longleftrightarrow \forall \sigma.\ A = A \cdot \sigma$$

Nonstrict and strict generalization are defined as

$$\text{generalizes } A\ B \longleftrightarrow \exists \sigma.\ A \cdot \sigma = B$$
$$\text{strictly\_generalizes } A\ B \longleftrightarrow \text{generalizes } A\ B \wedge \neg\ \text{generalizes } B\ A$$

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

$$\text{subsumes } C\ D \longleftrightarrow \exists \sigma.\ C \cdot \sigma \subseteq D$$
$$\text{strictly\_subsumes } C\ D \longleftrightarrow \text{subsumes } C\ D \wedge \neg\ \text{subsumes } D\ C$$

Unifiers and MGUs are characterized as follows, where $\mathcal{A} :: 'a\ set$ represents a unification constraint $A_1 \stackrel{?}{=} \cdots \stackrel{?}{=} A_k$ and $\mathcal{S} :: 'a\ set\ set$ represents a set of unification constraints:

$$\text{is\_unifier } \sigma\ \mathcal{A} \longleftrightarrow |\mathcal{A} \cdot \sigma| \leq 1$$
$$\text{is\_unifiers } \sigma\ \mathcal{S} \longleftrightarrow \forall \mathcal{A} \in \mathcal{S}.\ \text{is\_unifier } \sigma\ \mathcal{A}$$
$$\text{is\_mgu } \sigma\ \mathcal{S} \longleftrightarrow \text{is\_unifiers } \sigma\ \mathcal{S} \wedge (\forall \tau.\ \text{is\_unifiers } \tau\ \mathcal{S} \Longrightarrow \exists \gamma.\ \tau = \sigma \circ \gamma)$$

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:

locale *substitution* = *substitution_ops* +
  fixes
    renamings_apart :: $'a\ clause\ list \Rightarrow 's\ list$ **and**
    atm_of_atms :: $'a\ list \Rightarrow 'a$
  assumes
    $A \cdot \text{id} = A$ **and**
    $A \cdot (\sigma \circ \tau) = (A \cdot \sigma) \cdot \tau$ **and**
    $(\forall A.\ A \cdot \sigma = A \cdot \tau) \Longrightarrow \sigma = \tau$ **and**
    $\text{is\_ground\_cls } (C \cdot \sigma) \Longrightarrow \exists \tau.\ \text{is\_ground } \tau \wedge C \cdot \tau = C \cdot \sigma$ **and**
    wfP strictly_generalizes **and**
    $|\text{renamings\_apart } Cs| = |Cs|$ **and**
    $\rho \in \text{renamings\_apart } Cs \Longrightarrow \text{is\_renaming } \rho$ **and**

var_disjoint ($Cs \cdot$ renamings_apart Cs) **and**
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 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 σ given a set of unification constraints. If a unifier exists, it returns Some σ; 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. In this paper, we disable selection and hence only need to consider the binary case, which 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

$$\frac{C \vee A \vee \cdots \vee A \quad \neg A \vee D}{C \vee D}$$

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; $\neg A \vee B$ and $B \vee \neg A$ are the same clause.

For first-order logic, the order on atoms $>$ 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 nonground version of the calculus consists of the single inference 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 $A_1 \stackrel{?}{=} \cdots \stackrel{?}{=} A_k \stackrel{?}{=} A$, each $A_i \cdot \sigma$ is strictly $\succ$-maximal with respect to the atoms in $C \cdot \sigma$, and $A \cdot \sigma$ is $\succ$-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 $\mathcal{D}$ 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 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 $\mathcal{D}$ *saturated upto redundancy*, formally 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. It builds on the first-order ordered resolution rule and a redundancy criterion. The redundant clauses are those that are tautological (i.e., clauses of the form $C \vee A \vee \neg A$) and those that are subsumed by another clause in the working set—for example, the clauses B and $A \vee B$ are both subsumed if the working set already contains B. Furthermore, RP takes advantage of subsumption resolution, which can be expressed as an inference rule:

$$\frac{D \vee L \quad -(L \vee C \vee D) \cdot \sigma}{(C \vee D) \cdot \sigma}$$

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

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

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 $\rightsquigarrow$ :: *'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 $D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \cup \{C \uplus \{L\}\}) \rightsquigarrow_7 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow_8 (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$
| $(\{\}, \mathcal{P} \cup \{C\}, \mathcal{O}) \rightsquigarrow_9$ (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 set (or multiset) X under function f, infers_between $\mathcal{O}\ C$ calculates all the inferences whose premises are a subset of $\mathcal{O} \cup \{C\}$ that contains C, and reduces $D\ C\ L \Longleftrightarrow \exists D'\ L'\ \sigma.\ D = D' \uplus \{L'\} \wedge -L = L' \cdot \sigma \wedge D' \cdot \sigma \subseteq C$.

Example 4.1. There are many ways to derive $\bot$ from the unsatisfiable clause set $\{\mathrm{p}(x), \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\}$. 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.

$$
\begin{aligned}
&(\{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\},\ \{\}) \\
\rightsquigarrow_8\ &(\{\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\mathrm{p}(x)\},\ \{\}) \\
\rightsquigarrow_8\ &(\{\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\}) \\
\rightsquigarrow_9\ &(\{\},\ \{\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\mathrm{p}(x)\}) \\
\rightsquigarrow_9\ &(\{\neg\mathrm{p}(\mathrm{a})\},\ \{\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\}) \\
\rightsquigarrow_8\ &(\{\},\ \{\neg\mathrm{p}(\mathrm{a})\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\}) \\
\rightsquigarrow_9\ &(\{\bot\},\ \{\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}),\ \neg\mathrm{p}(\mathrm{a})\}) \\
\rightsquigarrow_8\ &(\{\},\ \{\bot\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}),\ \neg\mathrm{p}(\mathrm{a})\}) \\
\rightsquigarrow_9\ &(\{\},\ \{\},\ \{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}),\ \neg\mathrm{p}(\mathrm{a}),\ \bot\})
\end{aligned}
\qquad
\begin{aligned}
&(\{\mathrm{p}(x),\ \neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\},\ \{\}) \\
\rightsquigarrow_8\ &(\{\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b})\},\ \{\mathrm{p}(x)\},\ \{\}) \\
\rightsquigarrow_5\ &(\{\neg\mathrm{p}(\mathrm{b})\},\ \{\mathrm{p}(x)\},\ \{\}) \\
\rightsquigarrow_5\ &(\{\bot\},\ \{\mathrm{p}(x)\},\ \{\}) \\
\rightsquigarrow_3\ &(\{\bot\},\ \{\},\ \{\}) \\
\rightsquigarrow_8\ &(\{\},\ \{\bot\},\ \{\}) \\
\rightsquigarrow_9\ &(\{\},\ \{\},\ \{\bot\})
\end{aligned}
$$

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

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

We can leave $\neg\mathrm{p}(\mathrm{a},\mathrm{a})$ 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 employing a fair strategy for moving clauses from $\mathcal{P}$ to $\mathcal{O}$.

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 $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \mathcal{S}_2 \rightsquigarrow \cdots$ 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* ⇒ *'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *bool* **where**
  chain *R* (LCons *x* LNil)
| chain *R xs* ∧ *R x* (lhd *xs*) ⟹ 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 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 $\mathcal{S}s$. For the rest of this section, we assume that $\mathcal{S}s$ is a derivation.

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

theorem *RP_sound*:
$\{\} \in \mathsf{Liminf}\ \mathcal{S}s \Longrightarrow \neg\ \mathsf{satisfiable}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s))$

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

theorem *RP_model*:
$\mathcal{S} \rightsquigarrow \mathcal{S}' \Longrightarrow (I \models \mathsf{grounding\_of}\ \mathcal{S}' \Longleftrightarrow I \models \mathsf{grounding\_of}\ \mathcal{S})$

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

theorem *RP_saturated_if_fair*:
$\mathsf{fair}\ \mathcal{S}s \Longrightarrow \mathsf{saturated\_upto}\ (\mathsf{Liminf}\ (\mathsf{grounding\_of}\ \mathcal{S}s))$

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*:
$\mathsf{fair}\ \mathcal{S}s \wedge \neg\ \mathsf{satisfiable}\ (\mathsf{grounding\_of}\ (\mathsf{lhd}\ \mathcal{S}s)) \Longrightarrow \{\} \in \mathcal{O}\_\mathsf{of}\ (\mathsf{Liminf}\ \mathcal{S}s)$

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 $\mathcal{P}$ 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 $\mathsf{full\_chain}\ (\rightsquigarrow_\mathsf{w})\ \mathcal{S}s$, where the full_chain predicate is defined coinductively as

coinductive $\mathsf{full\_chain} :: ('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathsf{llist} \Rightarrow \mathit{bool}$ **where**
$(\forall y.\ \neg\ R\ x\ y) \Longrightarrow \mathsf{full\_chain}\ R\ (\mathsf{LCons}\ x\ \mathsf{LNil})$
$\mid\ \mathsf{full\_chain}\ R\ xs \wedge R\ x\ (\mathsf{lhd}\ xs) \Longrightarrow \mathsf{full\_chain}\ R\ (\mathsf{LCons}\ x\ xs)$

and characterized by the equivalence

lemma *full_chain_iff_chain*:
$\mathsf{full\_chain}\ R\ xs \Longleftrightarrow \mathsf{chain}\ R\ xs \wedge (\mathsf{lfinite}\ xs \Longrightarrow \forall y.\ \neg\ R\ (\mathsf{llast}\ xs)\ y)$

For the rest of this section, we fix a full chain $\mathcal{S}s$ such that $\mathcal{P}\_\mathsf{of}\ (\mathsf{lhd}\ \mathcal{S}s) = \mathcal{O}\_\mathsf{of}\ (\mathsf{lhd}\ \mathcal{S}s) = \{\}$.

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 $\mathsf{fair}\ \mathcal{S}s$.

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 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 $i < j \Longrightarrow \text{weight}\ (C, i) < \text{weight}\ (C, j)$

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**
$\text{Neg}\ A \in C \wedge \text{Pos}\ A \in C \Longrightarrow (\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w}1} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ (\mathcal{P} \uplus \mathcal{O}) \wedge \text{subsumes}\ D\ C \Longrightarrow (\mathcal{N} + \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w}2} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ \mathcal{N} \wedge C \in \text{fst}\ `\ \mathcal{P} \wedge \text{strictly\_subsumes}\ D\ C \Longrightarrow$
$(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w}3} (\mathcal{N}, \{(E, k) \in \mathcal{P}.\ E \neq C\}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ \mathcal{N} \wedge \text{strictly\_subsumes}\ D\ C \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C, i)\}, t) \rightsquigarrow_{\text{w}4} (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ (\mathcal{P} \uplus \mathcal{O}) \wedge \text{reduces}\ D\ C\ L \Longrightarrow$
$(\mathcal{N} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w}5} (\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ \mathcal{N} \wedge \text{reduces}\ D\ C\ L \wedge (\forall j.\ (C \uplus \{L\}, j) \in \mathcal{P} \Longrightarrow j \leq i) \Longrightarrow$
$(\mathcal{N}, \mathcal{P} \uplus \{(C \uplus \{L\}, i)\}, \mathcal{O}, t) \rightsquigarrow_{\text{w}6} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
$| D \in \text{fst}\ `\ \mathcal{N} \wedge \text{reduces}\ D\ C\ L \Longrightarrow (\mathcal{N}, \mathcal{P}, \mathcal{O} \uplus \{(C \uplus \{L\}, i)\}, t) \rightsquigarrow_{\text{w}7} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
$| (\mathcal{N} \uplus \{(C, i)\}, \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w}8} (\mathcal{N}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t)$
$| (\forall (D, j) \in \mathcal{P}.\ \text{weight}\ (C, i) \leq \text{weight}\ (D, j)) \wedge$
$\mathcal{N} = \text{mset\_set}\ ((\lambda D.\ (D, t))\ `\ \text{concl\_of}\ `\ \text{infers\_between}\ (\text{set\_mset}\ (\text{fst}\ `\ \mathcal{O}))\ C) \Longrightarrow$
$(\{\}, \mathcal{P} \uplus \{(C, i)\}, \mathcal{O}, t) \rightsquigarrow_{\text{w}9} (\mathcal{N}, \{(D, j) \in \mathcal{P}.\ D \neq C\}, \mathcal{O} \uplus \{(C, i)\}, t + 1)$

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. 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 $(\mathcal{N} \cup \{C\}, \mathcal{P}, \mathcal{O}) \rightsquigarrow (\mathcal{N}, \mathcal{P} \cup \{C\}, \mathcal{O})$ where $C \in \mathcal{N} \cap \mathcal{P}$. The use of multisets and $\uplus$ 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 reduction alters 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 a state component) 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

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

To prevent this behavior, 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, which is proved by case distinction on the rules:

lemma *preserve_min_P*:
$\mathcal{S} \rightsquigarrow_\mathsf{w} \mathcal{S}' \wedge (C,i) \in \mathcal{P}\_\text{of}\ \mathcal{S} \wedge (\forall k.\ (C,k) \in \mathcal{P}\_\text{of}\ \mathcal{S} \Longrightarrow k \geq i) \wedge C \in \text{fst}\ `\ \mathcal{P}\_\text{of}\ \mathcal{S}' \Longrightarrow (C,i) \in \mathcal{P}\_\text{of}\ \mathcal{S}'$

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
| | :: *'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 $|C| = \sum_{A\,:\,A \in C \vee \neg A \in C} |A|$. 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. 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 clarify fine theoretical points.

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

$$\begin{aligned}
&(\{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, \{\}, \{\}, 1)\\
\rightsquigarrow_{\mathrm{w8}}\ &(\{(\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, \{(\mathrm{p}(x), 0)\}, \{\}, 1)\\
\rightsquigarrow_{\mathrm{w8}}\ &(\{\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, \{\}, 1)\\
\rightsquigarrow_{\mathrm{w9}}\ &(\{\}, \{(\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, \{(\mathrm{p}(x), 0)\}, 2)\\
\rightsquigarrow_{\mathrm{w9}}\ &(\{(\neg\mathrm{p}(\mathrm{a}), 2)\}, \{\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, 3)\\
\rightsquigarrow_{\mathrm{w8}}\ &(\{\}, \{(\neg\mathrm{p}(\mathrm{a}), 2)\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0)\}, 3)\\
\rightsquigarrow_{\mathrm{w9}}\ &(\{(\bot, 3)\}, \{\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0), (\neg\mathrm{p}(\mathrm{a}), 2)\}, 4)\\
\rightsquigarrow_{\mathrm{w8}}\ &(\{\}, \{(\bot, 3)\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0), (\neg\mathrm{p}(\mathrm{a}), 2)\}, 4)\\
\rightsquigarrow_{\mathrm{w9}}\ &(\{\}, \{\}, \{(\mathrm{p}(x), 0), (\neg\mathrm{p}(\mathrm{a}) \vee \neg\mathrm{p}(\mathrm{b}), 0), (\neg\mathrm{p}(\mathrm{a}), 2), (\bot, 3)\}, 5)
\end{aligned}$$

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

5.2 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*, without timestamps. Formally:

lemma *weighted_RP_imp_RP*:
$\mathcal{S} \rightsquigarrow_\mathrm{w} \mathcal{S}' \Longrightarrow \mathsf{state\_of}\ \mathcal{S} \rightsquigarrow \mathsf{state\_of}\ \mathcal{S}'$

The proof is by straightforward induction on the introduction rules of RP_w, with one difficult case. Inference computation (rule 9) converts a set to a finite multiset using mset_set. 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 infers_between:

lemma *finite_ord_FO_resolution_inferences_between*:
$\mathsf{finite}\ \mathcal{D} \Longrightarrow \mathsf{finite}\ (\mathsf{infers\_between}\ \mathcal{D}\ C)$

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 $CAA = C \vee A_1 \vee \cdots \vee A_k$ 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 returned by infers_between is finite, assuming $\mathcal{D}$ 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 (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 $\mathcal{A} = \bigcup_{D \in \mathcal{DC}} \mathsf{atms\_of}\ D$ and $\mathcal{AA} = \{\mathcal{B} \mid \mathsf{set\_mset}\ \mathcal{B} \subseteq \mathcal{A} \wedge |\mathcal{B}| \leq n\}$. Then all inferences between $\mathcal{D}$ and C belong to $\mathcal{DC} \times \mathcal{DC} \times \mathcal{AA} \times \mathcal{A}$, 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:

theorem *weighted_RP_model*:
$\mathcal{S} \rightsquigarrow_\mathrm{w} \mathcal{S}' \Longrightarrow (I \models \mathsf{grounding\_of}\ \mathcal{S}' \Longleftrightarrow I \models \mathsf{grounding\_of}\ \mathcal{S})$

Completeness is considerably more difficult. We first show that the use of timestamps ensures that all full RP_w derivations are fair. 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 $\mathcal{N}$ 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 $\mathrm{RP_w}$ 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 natural numbers to pairs, which is a well-founded relation. The measure is

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

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 $\mathcal{N}$.

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*:
$\mathcal{S} \rightsquigarrow_{\mathrm{w}} \mathcal{S}' \wedge (C, _) \in \mathcal{N}$_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.

What about the case where a clause C is stuck in $\mathcal{P}$? Lemma *preserve_min_P* (Section 5.1) states that in any step, either all copies of a clause $C \in \mathcal{P}$ are removed or the one with minimum timestamp is preserved. It follows that 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_in_P*:
$C \in$ 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 $(\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \equiv$
$\big($sum $((\lambda(C, \_).\ |C| + 1)$ ‘ $\{Di \in \mathcal{N} \uplus \mathcal{P} \uplus \mathcal{O} \mid p\ Di\})$, $|\{Di \in \mathcal{N} \mid p\ Di\}|$, $|\{Di \in \mathcal{P} \mid p\ Di\}|\big)$

Notice that RP_filtered_measure $(\lambda\_.$ True) essentially amounts to RP_basic_measure. In the formalization, we use RP_filtered_measure $(\lambda\_.$ True) to avoid code 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 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 rules other than rule 9 followed by an application of rule 9: $(\rightsquigarrow^*_{\mathrm{w1–8}} \circ \rightsquigarrow_{\mathrm{w9}})^\omega$. 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 ≡ natLess <lex> natLess <lex> natLess

The measure RP_filtered_measure $(\lambda(\_, i).\ i \leq w)$ decreases in 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 the measure to ensure that it 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 $w\ \mathcal{S}$ ≡
$(w + 1 -$ 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*:
$\mathcal{S} \rightsquigarrow_w \mathcal{S}' \wedge Ci \in \mathcal{P}$_of $\mathcal{S} \Longrightarrow$
(RP_combined_measure (weight Ci) $\mathcal{S}'$, RP_combined_measure (weight Ci) $\mathcal{S}$)
∈ 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 fairness for all derivations starting with $\mathcal{P} = \mathcal{O} = \{\}$:

theorem *weighted_RP_fair*:
fair (lmap state_of $\mathcal{S}s$)

Since all derivations in RP_w are fair and its derivations are also derivations of RP, it is trivial to lift RP's saturation and completeness theorems, *RP_saturated_if_fair* and *RP_complete_if_fair*:

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

corollary *weighted_RP_complete*:
¬ satisfiable (grounding_of (lhd $\mathcal{S}s$)) $\Longrightarrow \{\} \in \mathcal{O}$_of (Liminf (lmap state_of $\mathcal{S}s$))

6 ELIMINATING NONDETERMINISM

The third refinement layer defines a functional program RP_d that embodies a specific rule application strategy, thereby eliminating the nondeterminism present in 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 $\mathrm{RP}_d\ (\mathcal{N}, \mathcal{P}, \mathcal{O}, t)$ **is**
  repeat forever
    if $\bot \in \mathcal{P} \uplus \mathcal{O}$ **then**
      return $\mathcal{P} \uplus \mathcal{O}$
    else if $N = P = \{\}$ **then**
      return $\mathcal{O}$
    else if $N = \{\}$ **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, $\mathcal{P} = \mathcal{O} = \{\}$, and an arbitrary timestamp t. The loop is loosely modeled after the proof procedure implemented in Vampire [Voronkov 2014, Section 3].

Instead of finite multisets, the actual $\mathsf{RP_d}$ 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 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
```

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

The prover is defined inside a locale that inherits *weighted_FO_resolution_prover_with_size_timestamp_factors*. The core function, $\mathsf{RP_d}$_step, performs a single iteration of the main loop. Here is the complete definition, excluding auxiliary functions:

```
fun RPd_step :: 'a dstate ⇒ 'a dstate where
  RPd_step (N, P, O, t) =
  if ∃Ci ∈ P @ O. fst Ci = [] then
    ([], [], remdups P @ O, t + |remdups P|)
  else
    (case N of
      [] ⇒
      (case P of
        [] ⇒ (N, P, O, t)
      | P0 # P' ⇒
        let
          (C, i) = select_min_weight_clause P0 P';
          N = map (λD. (D, t)) (remdups (resolve_rename C C
            @ concat (map (resolve_rename_either_way C ∘ fst) O)));
          P = filter (λ(D, j). mset D ≠ mset C) P;
          O = (C, i) # O;
          t = t + 1
        in
          (N, P, O, t))
    | (C, i) # N ⇒
      let
        C = reduce (map fst (P @ O)) [] C
```

```
in
  if C = [] then
    ([], [], [([], i)], t + 1)
  else if is_tautology C ∨ subsume (map fst (P @ O)) C then
    (N, P, O, t)
  else
    let
      P = reduce_all C P;
      (back_to_P, O) = reduce_all2 C O;
      P = back_to_P @ P;
      O = filter ((¬) ∘ strictly_subsume [C] ∘ fst) O;
      P = filter ((¬) ∘ strictly_subsume [C] ∘ fst) P;
      P = (C, i) # P
    in
      (N, P, O, t))
```

The code above relies on some nonexecutable constructs, such as the existential quantifier. The quantifier is unproblematic because it ranges over a finite set, but some of the auxiliary functions rely on infinite quantification. Notably, subsumption of D by C is defined as $\exists\sigma.\ C \cdot \sigma \subseteq D$ (Section 3), where σ ranges over all 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 σ using matching, instead of blindly enumerating all possibilities.

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

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

Since there are no guarantees that the recursion will terminate, we cannot introduce the function using the **fun** command [Krauss 2006], which is restricted to well-founded recursion. Instead, we use **partial_function** (*option*) [Krauss 2010], 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*:
  $\mathsf{RP_d}\ \mathcal{S}$ = Some $R \Longrightarrow \exists \mathcal{S}'\ k.\ \mathsf{RP_d\_step}^k\ \mathcal{S} = \mathcal{S}' \wedge$ is_final $\mathcal{S}' \wedge R =$ map fst ($\mathcal{O}$_of $\mathcal{S}'$)

6.2 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*:
  ¬ is_final $\mathcal{S} \Longrightarrow$ wstate_of $\mathcal{S} \rightsquigarrow_{\mathsf{w}}^{+}$ 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 the definition of $\mathsf{RP_d\_step}$:

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

By induction on |remdups $\mathcal{P}$|, there must exist a derivation of the form

$$\begin{aligned} & \text{wstate\_of } (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow^*_{\text{w2}} \ & \text{wstate\_of } ([], \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\text{w9}} \ & \text{wstate\_of } (\mathcal{N}', \mathcal{P}', (C, i) \# \mathcal{O}, t + 1) \\ \rightsquigarrow^*_{\text{w}} \ & \text{wstate\_of } ([], [], \text{remdups } \mathcal{P}' @ \mathcal{O}, t + |\text{remdups } \mathcal{P}'|) \end{aligned}$$

for $\mathcal{P}'$ = filter ($\lambda(D, j)$. mset $D \neq$ mset C) $\mathcal{P}$ 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

$$\text{wstate\_of } ([], \mathcal{P}, \mathcal{O}, t) \rightsquigarrow_{\text{w9}} \text{wstate\_of } (\mathcal{N}', \mathcal{P}', (C, i) \# \mathcal{O}, t + 1)$$

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

$$\begin{aligned} \mathcal{N}' &= \text{map } (\lambda D.\ (D, t)) \text{ (remdups (resolve\_rename } C\ C \\ &\qquad @ \text{ concat (map (resolve\_rename\_either\_way } C \circ \text{fst) } \mathcal{O}))) \\ \mathcal{P}' &= \text{filter } (\lambda(D, j).\ \text{mset } D \neq \text{mset } C)\ \mathcal{P} \end{aligned}$$

The main proof obligation is that $\mathcal{N}'$, converted to multisets, equals the multiset mset_set ((λD. (D, t)) ‘ concl_of ‘ infers_between (set_mset (fst ‘ $\mathcal{O}$)) C) specified in rule $\rightsquigarrow_{\text{w9}}$. The distance between the functional program and its mathematical specification is at its greatest here. The proof is tedious but straightforward.

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

$$\begin{aligned} & \text{wstate\_of } ((C, i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow^*_{\text{w5}} \ & \text{wstate\_of } (([], i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow^*_{\text{w3}} \ & \text{wstate\_of } (([], i) \# \mathcal{N}', [], \mathcal{O}, t) \\ \rightsquigarrow^*_{\text{w4}} \ & \text{wstate\_of } (([], i) \# \mathcal{N}', [], [], t) \\ \rightsquigarrow_{\text{w8}} \ & \text{wstate\_of } (\mathcal{N}', [([], i)], [], t) \\ \rightsquigarrow^*_{\text{w2}} \ & \text{wstate\_of } ([], [([], i)], [], t) \\ \rightsquigarrow_{\text{w9}} \ & \text{wstate\_of } ([], [], [([], i)], t) \end{aligned}$$

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

$$\begin{aligned} & \text{wstate\_of } ((C, i) \# \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow^*_{\text{w5}} \ & \text{wstate\_of } ((C', i) \# \mathcal{N}, \mathcal{P}, \mathcal{O}, t) \\ \rightsquigarrow_{\text{w1,2}} \ & \text{wstate\_of } (\mathcal{N}, \mathcal{P}, \mathcal{O}, t) \end{aligned}$$

Otherwise:

$$\begin{aligned}
&\text{wstate_of } ((C, i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t)\\
\rightsquigarrow_{\text{w5}}^{*}\ &\text{wstate_of } ((C', i) \# \mathcal{N}', \mathcal{P}, \mathcal{O}, t)\\
\rightsquigarrow_{\text{w6}}^{*}\ &\text{wstate_of } ((C', i) \# \mathcal{N}', \mathcal{P}', \mathcal{O}, t)\\
\rightsquigarrow_{\text{w7}}^{*}\ &\text{wstate_of } ((C', i) \# \mathcal{N}', \text{back_to_}\mathcal{P} \mathbin{@} \mathcal{P}', \mathcal{O}', t)\\
\rightsquigarrow_{\text{w4}}^{*}\ &\text{wstate_of } ((C', i) \# \mathcal{N}', \text{back_to_}\mathcal{P} \mathbin{@} \mathcal{P}', \mathcal{O}'', t)\\
\rightsquigarrow_{\text{w3}}^{*}\ &\text{wstate_of } ((C', i) \# \mathcal{N}', \mathcal{P}'', \mathcal{O}'', t)\\
\rightsquigarrow_{\text{w8}}\ &\text{wstate_of } (\mathcal{N}', (C', i) \# \mathcal{P}'', \mathcal{O}'', t)
\end{aligned}$$

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

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

lemma *final_deterministic_RP_step*:
is_final $\mathcal{S} \Longrightarrow \text{RP}_\text{d}\text{\_step } \mathcal{S} = \mathcal{S}$

6.3 Soundness and Completeness Proofs

Let $\mathcal{S}_0 = (\mathcal{N}_0, [], [], t_0)$ be an arbitrary initial state. For RP_d, soundness means that whenever $\text{RP}_\text{d}\ \mathcal{S}_0$ 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. Note that for satisfiable clause sets $\mathcal{N}_0$, both termination and divergence are possible.

To lift soundness and completeness results from RP_w to RP_d, we first define $\mathcal{S}s$ as a full chain of nontrivial RP_d steps starting from $\mathcal{S}_0$. Formally, we let $\mathcal{S}s$ = 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 (RP_d_step $\mathcal{S}$))

Based on $\mathcal{S}s$, we let $w\mathcal{S}s$ = lmap wstate_of $\mathcal{S}s$ and note that $w\mathcal{S}s$ is a full chain of "big" $\rightsquigarrow_\text{w}^+$ steps. Using a lemma that will be proved in Section 6.4, we obtain a full chain $ssw\mathcal{S}s$ of "small" $\rightsquigarrow_\text{w}$ steps. 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 lclause list*
assumes $\text{RP}_\text{d}\ \mathcal{S}_0$ = Some R

The definition of RP_d, using **partial_function**, gives us an induction rule restricted to the case where 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)

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*:
¬ satisfiable (grounding_of $\mathcal{N}_0$) $\longleftrightarrow$ {} $\in R$

Completeness is proved in a separate nameless locale that assumes nontermination: $\mathsf{RP_d}\ \mathcal{S}_0 =$ 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 ¬ satisfiable (grounded_of $\mathcal{N}_0$). Hence, by *weighted_RP_complete* we have {} $\in \mathcal{O}$_of *sswSs*. It is easy to show that *sswSs*'s limit is a subset of *wSs*'s limit; hence {} $\in \mathcal{O}$_of *wSs*. This implies the existence of a natural number k such that {} $\in \mathcal{O}$_of (lnth *wSs* k). Hence {} $\in \mathcal{O}$_of ($\mathsf{RP_d}$_step^k $\mathcal{S}_0$). However, by an induction on k, we can show that $\mathsf{RP_d}$ must terminate after at most k iterations, contradicting the assumption that $\mathsf{RP_d}$ diverges.

6.4 A Coinductive Puzzle

A single "big" step of the deterministic prover $\mathsf{RP_d}$ may consist of multiple "small" steps of the weighted prover $\mathsf{RP_w}$. To transfer the results from $\mathsf{RP_w}$ to $\mathsf{RP_d}$, we must expand $\mathsf{RP_d}$'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 desired property is formalized 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 the function $+\!\!+$, 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. Formally: *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 (i.e., ¬ lfinite *xs* $\wedge$ ¬ lfinite *ys* $\Longrightarrow$ llast *xs* = llast *ys*).

To prove *chain_tranclp_imp_exists_chain*, 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

$$\begin{array}{l} \quad \mathsf{LCons}\ x\ (\mathsf{LCons}\ y\ ys) \Rightarrow \mathsf{LCons}\ x\ (\mathsf{pick}\ R\ x\ y \mathbin{+\!\!+} \mathsf{wit}\ R\ xs) \\ \mid \_ \Rightarrow xs) \end{array}$$

Here pick $R\ x\ y$ returns an arbitrary finite list of R-related intermediate states connecting the R^+-related states x and y. Formally,

$$\mathsf{pick}\ R\ x\ y = \mathsf{SOME}\ zs.\ \mathsf{chain}\ R\ (\mathsf{llist\_of}\ (x \#\ zs\ @\ [y]))$$

where llist_of converts finite lists into lazy list and SOME is Hilbert's choice operator. Thus, pick satisfies the characteristic property $R^+\ x\ y \Longrightarrow \mathsf{chain}\ R\ (\mathsf{llist\_of}\ (x \#\ \mathsf{pick}\ R\ x\ y\ @\ [y]))$. The use of Hilbert choice makes pick, and wit, nonexecutable. This is acceptable because these constants are used only in the proofs and not in the actual 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 subtle. This syntactic structure of the definition is called *corecursive up to* $+\!\!+$. What ensures wit's productivity in the end is the fact that $+\!\!+$ does not removes any LCons constructors from its second arguments. A slightly weaker requirement, called *friendliness*, is supported by Isabelle's **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 *chain_tranclp_imp_exists_chain* are then discharged separately under the common assumption $\mathsf{chain}\ R^+\ xs$. In increasing difficulty: $\mathsf{lhd}\ (\mathsf{wit}\ R\ xs) = \mathsf{lhd}\ xs$ follows by simple rewriting. Next, $\mathsf{llast}\ (\mathsf{wit}\ R\ xs) = \mathsf{llast}\ xs$ requires an induction in the case of finite chains xs. For any infinite chain zs of type *'a llist*, llast zs is defined as a fixed but not further specified value of type *'a*. The properties $xs \sqsubseteq \mathsf{wit}\ R\ xs$ and $\mathsf{chain}\ R\ (\mathsf{wit}\ R\ xs)$ require a coinduction on $\sqsubseteq$ and chain, respectively. 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:

$$\frac{\forall xs.\ P\ xs \Longrightarrow (\exists z.\ xs = \mathsf{LCons}\ z\ \mathsf{Nil}) \vee (\exists z\ zs.\ xs = \mathsf{LCons}\ z\ zs \wedge P\ zs \wedge R\ z\ (\mathsf{lhd}\ zs))}{\forall xs.\ P\ xs \Longrightarrow \mathsf{chain}\ R\ xs} \qquad \frac{\forall xs.\ P\ xs \Longrightarrow (\exists z.\ xs = \mathsf{LCons}\ z\ \mathsf{Nil}) \vee (\exists z\ zs\ ys.\ xs = \mathsf{LCons}\ z\ (ys \mathbin{+\!\!+} zs) \wedge P\ zs \wedge \mathsf{chain}\ R\ (z \#\ ys\ @\ [\mathsf{lhd}\ zs]))}{\forall xs.\ P\ xs \Longrightarrow \mathsf{chain}\ R\ xs}$$

The property *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 $\mathsf{RP_d}$ 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 λ-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:

datatype $('f, 'v)$ *term* =
  Var $'v$
| Fun $'f$ $(('f, 'v)$ *term list*$)$

To simplify notation, in this paper we fix $'f = 'v = nat$ and work with the monomorphic type $('f, 'v)$ *term*, which we abbreviate by *term*. In the formalization, polymorphic types are used whenever possible. IsaFoR also define the standard monadic term substitution $\cdot :: term \Rightarrow ('v \Rightarrow term) \Rightarrow term$ and a function unify $:: (term \times term)\ list \Rightarrow lsubst \Rightarrow lsubst$, 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 about the defined functions, including the correctness of unify and the well-foundedness of strict term generalization $> :: term \Rightarrow term \Rightarrow bool$ defined by $t > s \longleftrightarrow (\exists \sigma.\ s \cdot \sigma = t) \wedge (\nexists \sigma.\ t \cdot \sigma = s)$.

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 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 $term\ set\ set \Rightarrow ('v \Rightarrow term)\ option$. 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 renamings_apart was not present in IsaFoR. We supply this definition:

fun renamings_apart :: *term clause list* $\Rightarrow ('v \Rightarrow term)\ list$ **where**
  renamings_apart [] = []
| renamings_apart $(C \# Cs)$ =
  let
    σs = renamings_apart Cs;
    $\sigma = \lambda v.\ v + \max\ (\{0\} \cup$ vars_clause_list $(Cs \cdot \sigma s)) + 1$
  in $\sigma \# \sigma s$

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 further 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. Our approach is to instantiate $\succ$ 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 $\succ$ 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 $\succ$).

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 $\succ$. This instantiation propagates to subsequent definitions, such as sorting or computing the minimum. To use a different order for sorting, we must 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 concerns clause subsumption. Its mathematical definition, subsumes $C\ D \longleftrightarrow \exists\sigma.\ C \cdot \sigma \subseteq D$, involves an infinite quantification ranging over substitutions.

The problem of deciding whether such a substitution exists is 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:

```
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, *osubst* abbreviates $'v \Rightarrow$ *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 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 unification and matching procedures and is inherited by our subsumes_list.

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

Clause subsumption is a hot spot in a resolution prover. This has lead to the empirical studies of various heuristics to improve on the naive exhaustive search [Schulz 2013a; Tammet 1998]. Following Tammet [1998], we implement a simple but useful heuristic that often reduces the number of calls to match_term_list, match_term_list, which are linear in the input term sizes, 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. Accordingly, we sort our (list-represented) clauses with respect to a literal quasi-order (i.e., a transitive and reflexive relation) leq_lit such that

```
is_pos L = is_pos K ∧ match_term_list [(atm_of L, atm_of K)] σ = Some ρ ⟹ leq_lit L K
```

Any quasi-order satisfying this property can be used in a refinement of subsumes_list to remove too small literals (with respect to leq_lit), as highlighted in gray below:

```
fun subsumes_list′ :: term literal list ⇒ term literal list ⇒ osubst ⇒ 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 theorem subsumes_list *Ls Ks* ρ = subsumes_list′ (sort leq_lit *Ls*) *Ks* ρ allows the code generator to refine the unoptimized version. In our prover, we let leq_lit be a quasi-order that considers negative literals smaller than positive ones, that considers variables smaller than nonvariable terms, and that sorts terms according to a total order on their root symbols.

This refinement is a local optimization: It requires us to explicitly sort one of the input clauses. A more efficient but also more intrusive refinement would be to maintain the invariant that all clauses in the prover's state are sorted with respect to leq_lit. Sorting *Ls* for each invocation of clause 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.

7.3 The End Result

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

export_code prover **in** *SML* **module_name** *RP*

generates a Standard ML module containing the implementation of our prover in slightly more than 1000 lines of code, including dependencies. The generated module exports the ML function

```
val prover : ((nat, nat) term literal list * nat) list -> bool
```

Even though in Isabelle we have proved that for any unsatisfiable input prover will terminate and return False, the code generator guarantees only partial correctness of its output: If the generated program terminates on the ML input generated from the Isabelle term t and evaluates to the Boolean result b, the proposition prover $t = b$ is provable in Isabelle. (There is recent work towards providing stronger guarantees [Hupel and Nipkow 2018].) By soundness, we also know that the Boolean b indicates the satisfiability of the input clause set.

After working hard to obtain an executable prover, it would be a shame not to run it on some example. We selected benchmark MSC015 from the TPTP library [Sutcliffe 2017], a particularly challenging family Φ_n of first-order problems. Each problem consists of the following $n + 2$ clauses (2 unit clauses and n two-literal clauses):

$$\begin{aligned}
\neg\, \mathrm{p}(\mathrm{b}, \ldots, \mathrm{b}) \quad & \mathrm{p}(\mathrm{a}, \ldots, \mathrm{a}) \\
\neg\, \mathrm{p}(\mathrm{a}, \mathrm{b}, \ldots, \mathrm{b}) \vee\ & \mathrm{p}(\mathrm{b}, \mathrm{a}, \ldots, \mathrm{a}) \\
\neg\, \mathrm{p}(x_1, \mathrm{a}, \mathrm{b}, \ldots, \mathrm{b}) \vee\ & \mathrm{p}(x_1, \mathrm{b}, \mathrm{a}, \ldots, \mathrm{a}) \\
& \vdots \\
\neg\, \mathrm{p}(x_1, \ldots, x_{n-2}, \mathrm{a}, \mathrm{b}) \vee\ & \mathrm{p}(x_1, \ldots, x_{n-2}, \mathrm{b}, \mathrm{a}) \\
\neg\, \mathrm{p}(x_1, \ldots, x_{n-2}, x_{n-1}, \mathrm{a}) \vee\ & \mathrm{p}(x_1, \ldots, x_{n-2}, x_{n-1}, \mathrm{b})
\end{aligned}$$

A comment in the benchmark warns us that back in 2007, no prover could solve the Φ_{23} within an hour. Even in 2018, only one prover solves Φ_{22} within 300 s, and 4 provers solve Φ_{20} within 300 s. Our verified prover solves Φ_{20} in 100 s and Φ_{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] 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 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 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 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, can be used to 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 certify the proof. Isabelle's support for coinductive methods, including the **coinductive**, **codatatype**, and **corec** commands, helps reason about infinite processes. Locales are a useful abstraction for defining the refinement layers. And the libraries included in the Isabelle distribution, the *Archive of Formal Proofs*, and the third-party IsaFoR certainly saved us months of work.

The *Archive* also includes a refinement framework [Lammich 2013], which has been used in a separate effort 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 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 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 Ph.D. 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. 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 [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 beyond layer 1.

We built our verified prover on Schlichtkrull et al.'s [2018a,b] formalization of ordered resolution. Related efforts, developed using Isabelle/HOL, include Peltier's [2016] formalization of superposition

and Schlichtkrull's [2018] formalization of unordered resolution. However, these developments only cover 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 [2017] 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 [2005] verified first-order prover, based on a sequent calculus for first-order logic without full first-order terms but only variables. Kumar et al. [2016] formalized the soundness of a proof assistant for higher-order logic. Jensen et al. [2018] 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 [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, 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. 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 are now advocating this approach for their systems as well [Reger and Suda 2017]. 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 [2018a,b] abstract specification of an ordered resolution prover, 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.

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.

ACKNOWLEDGMENTS

Mathias Fleury, Andreas Halkjær From, and Jørgen Villadsen suggested many textual improvements. Blanchette has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

REFERENCES

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.

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.

Clemens Ballarin. 2014. Locales: A Module System for Mathematical Theories. *J. Autom. Reasoning* 52, 2 (2014), 123–153.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Springer.

Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. 2017. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In *FroCoS 2017 (LNCS)*, Clare Dixon and Marcelo Finger (Eds.), Vol. 10483. Springer, 3–21.

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.

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 (LNCS)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 111–140.

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

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda—a Functional Language with Dependent Types. In *TPHOLs 2009 (LNCS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 73–78.

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68.

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 (LNCS)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 220–236.

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.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS, Vol. 78. Springer.

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

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

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

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

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

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

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.

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.

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

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

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

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.

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

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

Leslie Lamport. 1995. How to Write a Proof. *Amer. Math. Monthly* 7, 102 (1995), 600–608.

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

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

Filip Marić. 2010. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science* 411, 50 (2010), 4333–4356. https://doi.org/10.1016/j.tcs.2010.09.014

R. Milner. 1984. The use of machines to assist in rigorous proof. *Phil. Trans. R. Soc. Lond. A* 312 (1984), 411–422.

Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics: With Isabelle/HOL*. Springer.

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

Andreas Nonnengart and Christoph Weidenbach. 2001. Computing Small Clause Normal Forms. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier, 335–367.

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.

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 (EPiC Series in Computing)*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.), Vol. 2. EasyChair, 1–11.

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

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

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

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

Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. 2018a. 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.

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

Stephan Schulz. 2013a. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Automated Reasoning and Mathematics—Essays in Memory of William W. McCune (LNCS)*, Maria Paola Bonacina and Mark E. Stickel (Eds.), Vol. 7788. Springer, 45–67.

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

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.

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

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.

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.

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

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

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

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

Makarius Wenzel. 2007. Isabelle/Isar—a Generic Framework for Human-Readable Proof Documents. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, Roman Matuszewski and Anna Zalewska (Eds.). Studies in Logic, Grammar, and Rhetoric, Vol. 10(23). University of Białystok.

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

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