# Formal Verification of Logical Calculi and Simulations in Isabelle/HOL

Dissertation
zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Martin Desharnais

Saarbrücken, 2024

**Tag des Kolloquiums:** 29. Januar 2025

**Dekan:** Prof. Dr. Roland Speicher

**Prüfungsausschuss:**

| | |
|---|---|
| Vorsitzender: | Prof. Dr. Gert Smolka |
| Akademischer Mitarbeiter: | Dr. Andreas Schmidt |
| Berichterstattende: | Prof. Dr. Jasmin Blanchette |
| | Dr. Martin Bromberger |
| | Dr. Nicolas Peltier |
| | Dr. Sophie Tourret |
| | Prof. Dr. Christoph Weidenbach |

# Abstract

This thesis describes my formalizations of three proof calculi: SCL(FOL), ground ordered resolution, and ground superposition. The main theorems formalized for each calculus are soundness (i.e., every formula derived from valid formulas is valid) and refutational completeness (i.e., if a formula is invalid, then the calculus can be used to derive a refutation). For SCL(FOL), another main theorem is that derived formulas are nonredundant (i.e., they are not "obvious" from the already known formulas).

Ground ordered resolution only has this last property when a suitable strategy is used. I re-proved and formalized a previously known result that a specific strategy for SCL(FOL) can simulate a specific strategy for ground ordered resolution and vice versa. This was carried out with a framework for simulation proofs that I developed.

Finally, I formalized three interpreters for dynamically typed bytecode languages. The first one serves as a baseline, and the other two add speculative optimizations. I proved that the first interpreter can simulate the second one and vice versa, and also that the second interpreter can simulate the third one and vice versa. This formalization was also done using my framework for simulation proofs.

All formalizations were carried out using the Isabelle/HOL proof assistant.

# Zusammenfassung

Diese Dissertation beschreibt meine Formalisierungen von drei Kalkülen: SCL(FOL), Geordnete Grundresolution, und Grundsuperposition. Die formalisierten Haupttheoreme sind Korrektheit (d.h., jede durch den Kalkül von gültigen Formeln abgeleitete Formel ist selbst gültig) und Widerspruchsvollständigkeit (d.h., aus einer unerfüllbaren Formel kann der Kalkül einen Widerspruch ableiten). Für SCL(FOL) ist ein weiteres Haupttheorem, dass jede abgeleitete Formel nicht redundant ist (d.h., sie folgt nicht „trivialerweise“ aus bereits bekannten Formeln).

Geordnete Grundresolution genießt diese letzte Eigenschaft nur dann, wenn eine geeignete Strategie Anwendung findet. Ich bewies das bereits bekannte Ergebnis erneut und formalisierte, dass eine spezifische Strategie für SCL(FOL) eine spezifische Strategie für Geordnete Grundresolution simulieren kann und umgekehrt. Dies wurde mithilfe eines von mir entwickelten Frameworks für Simulationsbeweise durchgeführt.

Abschließend formalisierte ich drei Interpreter für dynamisch typisierte Bytecodesprachen. Der Erste dient als Vergleichsbasis und die Anderen fügen spekulative Optimierungen hinzu. Ich bewies, dass der Erste den Zweiten simulieren kann und umgekehrt sowie dass der Zweite den Dritten simulieren kann und umgekehrt. Dies wurde ebenfalls mithilfe meines Frameworks für Simulationsbeweise durchgeführt.

Sämtliche Formalisierungen wurden mithilfe des Beweisassistenten Isabelle/HOL durchgeführt.

# Acknowledgments

First and foremost, I want to thank my supervisors Christoph Weidenbach, Jasmin Blanchette, Martin Bromberger, and Sophie Tourret.

Jasmin deserves special thanks because he lured me to Munich for a research internship during my bachelor and introduced me to interactive theorem proving. He helped to make myself at home in Munich and this positive experience is what convinced me to immigrate to Germany between my bachelor and master. We stayed in contact during my master and he cosupervised me when I started a PhD in Munich in the field of software security. After I realized that this was not for me, Jasmin supported me in my decision to change subject and supervisor.

At this point, Christoph gave me the opportunity to work in the field of automated reasoning in Saarbrücken. He shared his formidable knowledge of the field, provided me with a best-in-class work environment, and guided me to ambitious and exciting projects. I am immensely thankful to him for giving me a second chance at doing a PhD. From then on, Christoph, Jasmin, Martin, and Sophie teamed up to supervise me. Having four cosupervisors based in three different cities is uncommon, but they offered me high-quality supervision through regular virtual meetings and multiple in-person meetings. Special thanks to Sophie who often traveled from Nancy to Saarbrücken and with whom I had many fruitful discussions on the art of formalizing. I am also grateful to Martin Bromberger who helped and counseled me day in day out.

I also want to thank all my colleagues at the Max Planck Institute for informatics for the enjoyable and interesting discussions we had. Special thanks to Lorenz Leutgeb and Simon Schwarz for making me feel at home in a new city. Jennifer Müller also deserves special thanks for the hard work she is doing as secretary and pillar of the group.

Most of the chapters in this thesis are based on various publications. I want to thank all my coauthors for the fruitful collaborations. Special thanks to Balazs Toth with whom it was a pleasure to collaborate on the Isabelle/HOL formalization.

On a more personal note, I would like to thank my family for always supporting me, especially when I decided to immigrate to Germany. Thanks to my mother and father, Josée Gaboury and André Desharnais, my grand-mother, Claudette Drouin, and to my siblings, Danielle, Benoit, and Maxence.

Last but not least, I want to thank my fiancé, Kai Schäfer, for being such an amazing partner and for the support he provided me for so many years. We have been through thick and thin together in the last few years and it now feels like we can finally settle down. I also want to thank Kai's parents, Gisela Schäfer-Reitz and Jürgen Reitz, for offering me a second family in Germany.

# Contents

# Chapter 1

# Introduction

Logical reasoning is a process to convince oneself (or another person) of a mathematical statement. The statement is broken down in *premises* (a.k.a. assumptions) and *conclusion*. Starting with the premises, one repeatedly performs some "obvious" *inference steps* until the conclusion is reached. Which inference steps are considered "obvious" can greatly vary from one situation to the other and from one person to the other. The sequence of inference steps from the premises to the conclusion is then called a *proof* of the statement.

To check a proof, one needs only to consider each inference step in isolation and ensure that it is a "correct" inference step; there is no need to consider the statement as a whole. This process, when done correctly, can result in high confidence in the truth of the statement as each step of the proof checking is straightforward.

Consider the following exemplary claim and the accompanying proof of a simple statement about the sum of integers.

**Claim.** *The sum of two even integers is even.*

*Proof.* Let $a$ and $b$ be two integers. Assume that $a$ and $b$ are both even. Remember that an integer $n$ is even if there exists an integer $k_n$ such that $n = 2 \times k_n$ (i.e., if $n$ is divisible by two). Because $a$ and $b$ are both even, there must exist two integers $k_a$ and $k_b$ such that $a = 2 \times k_a$ and $b = 2 \times k_b$. The sum of $a$ and $b$, written $a + b$, is equal to $(2 \times k_a) + (2 \times k_b)$ by rewriting the above equalities, which is in turn equal to $2 \times (k_a + k_b)$ by elementary arithmetic. This corresponds exactly to the definition of an even integer, which means that the sum of $a$ and $b$ is even. □

The premises of the statement are that there are two integers and that both are even; the conclusion is that their sum is even. The level of details in the above proof (i.e., the "obvious" inference steps) was chosen such that only elementary knowledge of arithmetic (as usually thought in elementary school) is sufficient to check the proof. For more complex statements, another level of details could be chosen (e.g., to keep the proof at a manageable size).

Now consider the following exemplary claim and the accompanying putative proof of a more complex statement about the colors of apples.

**Claim.** *All apples are of the same color.*

*Proof.* We shall use mathematical induction w.r.t. the number of apples. Let us check the first inductive step—a set of one apple is a set of apples of the same color.

We assume now that (for a given $n$) all apples in an $n$-element set of apples are of the same color. Let us add a new apple to any $n$-element set. We have a $(n+1)$-element set. Now let us take away an apple from the set, but not the one we have just added. We get a $n$-element set of apples. From the inductive assumption, all apples in this set are of the same color. Accordingly, the apple we have added is of the same color as other ones. Now we can bring back the eliminated apple (which is obviously of the same color as the rest) and we get a $(n+1)$-element set of apples of the same color. By virtue of mathematical induction we have proved that all apples are of the same color. □

The above putative proof is much more complex; it requires knowledge of sets and mathematical induction, leaves some details implicit (e.g., that the set of apples is finite due to us being able to count their number), and requires to keep track of multiple sets, apples, and colors at the same time. Checking this putative proof requires so much attention and effort that many people, on first read, gloss over the fact that one inference step is incorrect.

The statement does not hold and the claim is false: Figure 1.1 shows a counterexample. The incorrect inference step is in the penultimate sentence, which states that the eliminated apple "is obviously of the same color as the rest". Behind this "obviously" hides the following reasoning: the eliminated apple has the same color as the other apples in the original $n$-element set (from the first application of the induction hypothesis), which have the same color as the added apple (from the second application of the induction hypothesis). By transitivity, the colors of the eliminated apple and of the added apple should be the same. But this does not hold if the original $n$-element set contains only one apple, as the set of other apples would be empty.

Consider the counterexample from Fig. 1.1. The inductive step could either start with the 1-element set consisting of the red apple and then add the green apple, or start with the 1-element set consisting of the green apple and then add the red apple. Without loss of generality, let us pick the former. We assume a 1-element set consisting of a red apple; from the induction hypothesis, all apples in this set are of the same color (i.e., red). We then add the green apple. We get a 2-element set consisting of a red and a green apple. We then remove the red apple. We get a 1-element set consisting of a green apple; from the induction hypothesis, all elements in this set are of the same color (i.e., green). Accordingly, the green apple is of the same color as other ones (i.e., all apples in the empty set). We now bring back the eliminated red apple, which has the same color as the other ones (i.e., all apples in the empty set). The "obvious" inference step does not work and we get a 2-element set of one green and one red apple, which are not of the same color.

This claim is presented under the name "Horses Paradox" by Łukowski in his book *Paradoxes* [81, Section 2.5], where he considers the color of horses instead of apples. The above incorrect proof is adapted from Łukowski's book.

In summary, the first exemplary claim shows that proof checking consists of repetitive small checks—and that these checks can be very simple—while the second exemplary claim shows that proof checking can be nontrivial—and that human are fallible. But when keeping an appropriate level of details, the checks can be kept simple—so simple that they can be automated.

Figure 1.1: A finite set of two apples of different colors.
Picture by Joanna Malinowska (CC0 1.0).

A *proof assistant* (a.k.a. an interactive theorem prover) is a piece of software that takes a proof as input, checks every inference step, and outputs whether the proof is correct or not. The input proof must use a syntax understandable by the proof assistant and the level of details must be so that the simple automated checks suffice. This level of details is however quite different to what people are used to produce and consume. A certain time of learning and adaptation is usually required. But in exchange for the extra time and effort, proof assistants provide the strongest possible confidence in the correctness of one's proofs.

One such proof assistant is Isabelle/HOL [85]. Isabelle is a generic proof assistant that can be specialized to support different logical formalisms and Isabelle/HOL is the specialization for classical higher-order logic. Isabelle supports an input language, called Isar [123], that allows to write structured proofs that are readable for both humans and machines. Isabelle also has excellent support for proof automation: some tools are for general proof search (e.g., the simp tactic for equational rewriting or Sledgehammer [89] for integrating external automated theorem provers) and other tools are for specialized theories (e.g., the order tactic [108] for orderings or the presburger tactic [33] for Presburger arithmetic).

The process of writing claims and proofs in such a way that a proof assistant can understand and check them is called *formalizing* (a.k.a. formal verification, or interactive theorem proving). When formalizing the first exemplary claim and its proof in Isabelle/HOL, Isabelle confirms that it was able to check all proof steps. When formalizing the second exemplary claim and its incorrect proof, Isabelle complains that it is unable to verify the incorrect inference step. Isabelle even finds and reports a counterexample equivalent to the one in Fig. 1.1.

In this thesis, I describe how I used Isabelle/HOL to formalize multiple results in the fields of automated reasoning and virtual machines.

The field of automated reasoning studies *automated theorem provers*, or simply *provers*. They are pieces of software that take a mathematical statement as input and search for a proof; they either terminate with a proof of the statement (i.e., the statement is true), or they terminate with a refutation of the statement (i.e., the statement is false), or they never terminate. Two important properties of provers are *soundness* (i.e., if the prover claims that statement is true, then it is really true, and analogously if the prover claims it is false) and *completeness* (i.e., the prover will eventually terminate with either a proof or a refutation). Completeness is a very

strong property that quickly becomes unachievable when we consider expressive logics and rich theories. A weaker but still useful property is *refutational completeness* (i.e., if the statement is false, then the prover will eventually terminate with a refutation).

In the field of programming languages, a *virtual machine* is a piece of software used to execute a program without having to compile it to machine code ahead of time. A virtual machine does not usually operate directly on the source code of the program but rather on some intermediate representation called a bytecode language: an instruction set specifically designed to be efficiently executed by the virtual machine. Two important properties of a virtual machine are its *correctness* (i.e., it executes the program according to the semantics of the bytecode language) and its run-time *performance* (i.e., how efficient the execution of the program is). To improve efficiency, a virtual machine usually *optimizes* the given program (i.e., rewrites it in a semantics-preserving way to improve the performance of its execution).

Both fields have rich metatheories that define models of the systems (i.e., of a prover or of a virtual machine), precisely state the properties, and provide proofs of the properties. But this is usually done on paper and we rely on fallible humans to check the proofs. Because these proofs are often very complex, there is always a risk that an error might be hiding behind some "obvious" proof step (as was the case in the second exemplary claim above). Formalizing these properties in a proof assistant gives us the strongest possible confidence in the metatheory on which provers and virtual machines are based. At a high level, my main contributions are the following.

- I formalized multiple results in the metatheories of automated theorem proving and virtual machines in Isabelle/HOL.
- I improved upon the paper proofs by replacing some vague notions or proof steps by precise definitions and proofs—sometimes a vague sentence can turn into hundreds of lines of proof.
- I discovered and fixed two bugs in the material I was formalizing.

The rest of this thesis is organized in three parts:

- Part I consists of Chapters 3 to 5; it presents my formalizations of the ground superposition and first-order SCL(FOL) calculi.
- Part II consists of Chapters 6 and 7; it presents my formalizations of a framework for simulation proofs and its application to a bisimulation between a strategy of SCL(FOL) and a strategy of ground ordered resolution.
- Part III consists of Chapter 8; it presents my formalization of bisimulation proofs between optimizing interpreters for bytecode languages.

A short description of these chapters and, in each chapter, of my main contributions follows. Because these formalization projects are the result of joint work with colleagues, this thesis is generally written using the first-person plural pronoun (i.e., "we"). In contrast, the first person singular (i.e., "I") emphasizes my personal contributions.

**Chapter 2: An Overview of Isabelle/HOL** The Isabelle/HOL proof assistant and its main features are briefly introduced. The syntax of Isabelle/HOL is also covered as well as the similarities and differences to the notation used in this thesis.

**Chapter 3: Preliminaries on First-Order Logic** We briefly present the most important notions relating to first-order logic without equality: terms, term contexts, atoms, literals, clauses, substitutions, models, entailment, and orderings. The concepts are well-known and this presentation aims to inform the reader of the exact definitions used (we often had to choose between alternative definitions) and to introduce the notation for the different concepts.

My main contributions are

- an Isabelle/HOL formalization of a reusable theory of abstract substitution [41]; it is based on the concept of monoid actions and provides many useful definitions and lemmas,
- an Isabelle/HOL formalization of reusable theories of minimal, strictly minimal, maximal, strictly maximal, least, and greatest elements in sets, finite sets, and finite multisets [42]; it is based on restricted orderings and provides predicates for the different concepts and many useful lemmas, and
- numerous contributions to the Isabelle distribution (e.g., to the theories of finite sets, multisets, and properties of binaries relations) and to preexisting entries of the *Archive of Formal Proofs* (e.g., to the sessions `First_Order_Terms` [107], `Ordered_Resolution_Prover` [99], and `Saturation_Framework_Extensions` [21]).

**Chapter 4: Ground Superposition** Superposition is an efficient proof calculus for reasoning about first-order logic with equality [4, 5] that is implemented in several automatic theorem provers [10, 34, 36, 50, 70, 101, 102]. It works by saturating the given set of clauses and is refutationally complete, meaning that if the set is inconsistent, the saturation will contain a contradiction. We restructured the completeness proof to cleanly separate the ground (i.e., variable-free) and nonground aspects, and we formalized the result in Isabelle/HOL.

This chapter is based on a conference paper coauthored with Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret [48]. My main contribution is the Isabelle/HOL formalization of ground superposition [47] presented in Section 4.4; it includes proofs of soundness and refutational completeness. The Isabelle/HOL formalization of the lifting of ground refutational completeness to nonground refutational completeness [47] summarized in Section 4.5 is by Balazs Toth.

**Chapter 5: SCL(FOL): Simple Clause Learning for First-Order Logic** SCL(FOL) is a relatively new proof calculus for reasoning about first-order logic without equality [26, 28, 29, 52, 72] that lifts a conflict-driven clause learning approach [12, 82] to first-order logic. We present an Isabelle/HOL formalization of the calculus. The main results are formal proofs of soundness, nonredundancy of learned clauses, termination, and refutational completeness. Compared with the existing unformalized version, the formalized calculus is simpler and more general, some results such as nonredundancy are stronger and some results such as nonsubsumption are new. We found one bug in a previously published version of the SCL Backtrack rule. Compared with related formalizations, we introduce a new technique for showing termination based on nonredundant clause learning.

This chapter is based on a conference paper coauthored with Martin Bromberger

and Christoph Weidenbach [25]. My main contributions are

- the Isabelle/HOL formalization [40] presented in Section 5.3,
- the simplification of the calculus,
- the discovery and fix of a bug in a previously published version of the SCL Backtrack rule that made it possible to learn a duplicate clause (which contradicted the pen-and-paper theorem of nonredundant clause learning),
- the strengthening of the soundness and nonredundancy theorems,
- the monotonically decreasing measuring function used in the termination proof,
- the new proofs of nonsubsumption of learned clauses and the existence of a bound for any unsatisfiable clause set, and
- the generalization of the nonredundancy, nonsubsumption, and termination results to any strategy respecting some constraint.

**Chapter 6: A Framework for Simulation Proofs** We present a generic framework for formalizing compiler transformations. Our framework leverages Isabelle/HOL's locales [8]—a module system for generic formalizations—to abstract over concrete languages and transformations. The framework thus enables us to state common definitions for language semantics, program behaviours, forward and backward simulations, and compilers. We provide generic operations, such as compiler composition, and prove general theorems, resulting in reusable proof components.

This chapter is based on two workshop papers coauthored with Stefan Brunthaler [44, 45]. My main contribution is the Isabelle/HOL formalization [38] presented in Section 6.3.

**Chapter 7: Simulation between SCL(FOL) and Ground Ordered Resolution** The SCL(FOL) proof calculus is known to be able to simulate the derivation of nonredundant clauses by the ground ordered resolution calculus [27]. We reuse the existing strategy for ground ordered resolution and present a new, simpler strategy for SCL(FOL). We prove a stronger bisimulation theorem between these two strategies (i.e., they both simulate each other). Our proof is modular: it consists of ten refinement steps focusing on different aspects of the two strategies. To reduce the proof burden, we provide a lemma that lifts a simulation to a bisimulation. We formalized this proof in Isabelle/HOL.

This chapter describes unpublished work coproduced with Martin Bromberger and Christoph Weidenbach. My main contributions are

- the lifting lemma (joint work with Martin Bromberger) presented in Section 7.4 and its Isabelle/HOL formalization [38, Theory Lifting_Simulation_To_Bisimulation],
- the Isabelle/HOL formalization [43] presented in Sections 7.5 to 7.7,
- the design of the refinement steps (joint work with Martin Bromberger),
- the design of the new SCL(FOL) strategy (joint work with Martin Bromberger), and
- the generalization of the nonredundancy, nonsubsumption, and termination results for SCL(FOL) to projectable strategies.

**Chapter 8: Optimizing Virtual Machines** The prevalence of dynamic languages is not commensurate with the security guarantees provided by their execution mechanisms. Consider the ubiquitous case of JavaScript: it runs everywhere and its complex just-in-time compilers produce code that is fast and, unfortunately, sometimes incorrect. We present an Isabelle/HOL formalization of an alternative execution model—optimizing interpreters [31, 32, 51, 120, 126]—and mechanically verify its correctness. Specifically, we formalize three simple bytecode languages. One serves as baseline and the two others implement advanced speculative optimizations similar to those used in just-in-time compilers. We prove semantics preservation between the baseline language and the optimized ones. As a result, our formalization provides a path towards unifying vital performance requirements with desirable security guarantees.

This chapter is based on a conference paper coauthored with Stefan Brunthaler [46]. My main contributions are

- the Isabelle/HOL formalization [39] presented in Sections 8.3 to 8.6, and
- the identification of the source of a bug in the original CPython prototype.

# Chapter 2

# An Overview of Isabelle/HOL

Isabelle is a generic proof assistant that can support several logics. It implements a *metalogic* (a.k.a. a *logical framework*), called Isabelle/Pure (Section 2.1), in which different *object logics* such as higher-order logic (Section 2.2) can be expressed. These object logics can then be used to formalize project-specific theories. Some of Isabelle's strengths are its input language (Section 2.3), which allows to write human-readable structured proofs, its strong support for proof automation (Section 2.4), and its module system for hierarchies of theories (Section 2.5).

In this thesis, we will usually strive to stay close to Isabelle's notation, but deviate from it in a number of ways to improve readability and to stay closer to mathematical conventions (Section 2.6).

## 2.1 Metalogic

Isabelle's metalogic [86], called Isabelle/Pure, is an intuitionistic fragment of higher-order logic (a.k.a. *simple type theory*) based on the typed $\lambda$-calculus [35].

The *types* consist of variables (e.g., $'a$, $'b$, $'c$), fully applied $n$-ary type constructors usually written in postfix notation (e.g., Isabelle/HOL has the nullary type *bool*, the unary type $'a$ *set*, and the binary type $(bool, 'a)$ *prod*), and total unary functions usually written using the right-associative infix $\Rightarrow$ notation (e.g., $'a \Rightarrow 'b$ for the function with domain $'a$ and codomain $'b$); some types have a special notation (e.g., Isabelle/HOL has the special infix notation $bool \times 'a$ for the type $(bool, 'a)$ *prod*). All types are inhabited. All functions are strictly speaking unary, but we usually say that a type without arrow is a nullary function and that a type is a $(n + 1)$-ary function if its codomain is an $n$-ary function (e.g., $'a$ is nullary, $'a \Rightarrow 'b$ is unary, $'a \Rightarrow 'b \Rightarrow 'c$ is binary, $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd$ is ternary).

The *terms* consist of variables (e.g., $x$, $y$, $z$ could be nullary variables, $f$, $g$, $h$ could be nonnullary variables), constants (e.g., $\mathsf{True}$, $\mathsf{zero}$ could be nullary constants, $\mathsf{min}$, $\mathsf{max}$ could be nonnullary constants), function applications usually written in a left-associative curry style (e.g., $f\ x\ y$ to apply the function variable $f$ to the variables $x$ and $y$, $\mathsf{max}\ \mathsf{zero}\ z$ to apply the constant function $\mathsf{max}$ to the constant $\mathsf{zero}$ and the variable $z$, $\mathsf{min}\ x\ (\mathsf{min}\ y\ z)$ to apply the constant function $\mathsf{min}$ to the variable $x$ and to the result of $\mathsf{min}\ y\ z$), and function expressions (e.g., $\lambda x.\ x$ for the identity unary function, $\lambda x\ y.\ x$ for the binary function that evaluates to the first argument and discards the second, $\lambda f\ g\ x.\ f\ x\ (g\ x)$ for the ternary function that evaluates to the

first argument applied to the third and to the result of applying the second argument to the third); some terms have a special notation (e.g., in Isabelle/HOL, $x \cup y$ is a special infix notation for $\mathsf{Set.union}\ x\ y$). The notation $t :: \tau$ expresses that the term $t$ has type $\tau$.

Metalogical *formulas* are terms of type *prop* (i.e., the type of propositions). The metalogical *operations* are implication, denoted by $\Longrightarrow :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$, universal quantification, denoted by $\bigwedge :: ('a \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}$, and equality, denoted by $\equiv :: 'a \Rightarrow 'a \Rightarrow \mathit{prop}$; all three operations have a special notation (e.g., the formula $\bigwedge x\ y\ z.\ x \equiv y \Longrightarrow y \equiv z \Longrightarrow x \equiv z$ expresses the transitivity of metalogical equality). The symbols for metaoperations were chosen to differ from the symbols of object logics such as classical higher-order logic (described in Section 2.2).

The metalogic supports features such as rank-1 parametric polymorphism and Haskell-style type classes.

## 2.2 Object Logic

The Isabelle/HOL proof assistant is the instantiation of Isabelle for classical higher-order logic. It first axiomatizes a type *bool* of Booleans. Object-level *formulas* are terms of type *bool*. Isabelle/HOL then either axiomatizes or defines its own object-level logical constants with special notations: truth ($\mathsf{True} :: \mathit{bool}$), falsehood ($\mathsf{False} :: \mathit{bool}$), negation ($\neg :: \mathit{bool} \Rightarrow \mathit{bool}$), conjunction ($\wedge :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$), disjunction ($\vee :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$), implication ($\longrightarrow :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$), equivalence ($\longleftrightarrow :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$), equality ($= :: 'a \Rightarrow 'a \Rightarrow \mathit{bool}$), universal quantification ($\forall :: ('a \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$), existential quantification ($\exists :: ('a \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$), definite description ($\mathsf{The} :: ('a \Rightarrow \mathit{bool}) \Rightarrow 'a$), and indefinite description ($\mathsf{Eps} :: ('a \Rightarrow \mathit{bool}) \Rightarrow 'a$). Isabelle/HOL also axiomatizes some classical properties (e.g., the law of excluded middle). It also offers many reusable theories built on top of the basic axioms and definitions (e.g., for sets, lists, arithmetic).

The object logic is embedded in the metalogic through the constant $\mathsf{Trueprop} :: \mathit{bool} \Rightarrow \mathit{prop}$, which expresses that an object-level formula is a true metalogical formula; it is usually left implicit and not displayed. The distinction between metalogic and object logic is very important in day-to-day formalization: they are treated differently syntactically and some features are only available in one or the other (e.g., the notation $\mathsf{hyp}[\mathtt{of}\ t]$ to instantiates the outermost metalogical universal quantifier in the formula $\mathsf{hyp}$ with the term $t$ does not work with an object-level universal quantifier). As a rule of thumb, metalogical universal quantification and implication are usually preferred because they are better supported by the input language.

## 2.3 Input Language

Isabelle's input language, called Isar [123], allows to write definitions and structured proofs that are readable for both humans and machines.

Consider the exemplary claim from Chapter 1 that the sum of even integers is even. To formalize this claim, we would first need to define what even numbers are. The following Isabelle/HOL code defines a predicate that identifies even numbers.

```
definition even :: ‹int ⇒ bool› where
  ‹⋀n. even n ≡ (∃k_n. n = 2 * k_n)›
```

The command **definition** introduces a new constant named even of type $int \Rightarrow bool$ and binds the formula $\bigwedge n.\ \mathsf{even}\ n \equiv (\exists k_n.\ n = 2 * k_n)$ to the name even_def. We can now use both to claim and prove the claim on the sum of even integers.

```
theorem even_plus_even_is_even:
  fixes a :: int and b :: int
  assumes ‹even a› and ‹even b›
  shows ‹even (a + b)›
proof -
  obtain k_a :: int where ‹a = 2 * k_a›
    using ‹even a› even_def by blast

  obtain k_b :: int where ‹b = 2 * k_b›
    using ‹even b› even_def by auto

  have ‹a + b = (2 * k_a) + (2 * k_b)›
    by (simp add: ‹a = 2 * k_a› ‹b = 2 * k_b›)

  also have ‹⋯ = 2 * (k_a + k_b)›
    by presburger

  finally show ‹even (a + b)›
    by (metis even_def)
qed
```

The command **theorem** expects a name, a formula, and a proof; it checks that the formula is syntactically correct and checks that the proof is a sequence of correct inference steps. It then binds the formula to the name.

The formula is written using human-readable syntax: the keyword **fixes** identifies universally quantified variables, the keyword **assumes** identifies the premises, and the keyword **shows** identifies the conclusion. This is syntactic sugar for the formula

$$\bigwedge a\ b.\ \mathsf{even}\ a \Longrightarrow \mathsf{even}\ b \Longrightarrow \mathsf{even}\ (a + b)$$

which gets bound to the name even_plus_even_is_even.

The proof (everything between **proof** and **qed**) is also written using a human-readable syntax. The first step provides an integer $k_a$ such that the formula $a = 2 * k_a$ is true from the proof of its existence; the proof of existence uses one of the two premises and the definition of even integers and is done by one unique step of a tactic called blast—a tactic corresponds to a an automated proof procedure. The second step is analogous. The third step states and proves an intermediate property: that $a + b$ is equal to $(2 * k_a) + (2 * k_b)$. The fourth step states and proves another intermediate property: that the $\ldots$, a shorthand for the right-hand side of the equality in the previous property, (i.e., $(2 * k_a) + (2 * k_b)$) is equal to $2 * (k_a + k_b)$. In the last step, the keyword **finally** joins the two previous intermediate properties by transitivity (i.e., $a + b = (2 * k_a) + (2 * k_b) = 2 * (k_a + k_b)$), after which the keyword **show** states the conclusion and is followed by a final proof step.

When formalizing, we enjoy a lot of freedom in how we define, state, and prove things. Each possibility has advantages and disadvantages. The previous proof was optimized for readability but is relatively long. Had we wanted to minimize the size of the proof, we could have replace the **proof**, **qed**, and everything in between by the following one-line proof:

**by** (metis assms distrib_left even_def)

The proof consists of a single call to the tactic metis parametrized with a list of lemmas: assms is a name to which Isabelle automatically binds the premises even $a$ and even $b$, distrib_left is a lemma that states that multiplication on the left of an addition distributes, even_def is the definition of our even predicate for even integers. This proof is much shorter but does not convey the reasoning behind the proof to a human reader as well as the longer proof. So in my own formalizations, I use shorter proofs for obvious theorems and lemmas, and long proofs when the details are interesting.

## 2.4 Proof Automation

Isabelle/HOL provides strong proof automation. In the example of Section 2.3, every proof step was done with a different tactic. They are (in order of appearance in the above proof):

- blast is a classical tableau prover.
- auto combines classical reasoning with simplification.
- simp performs simplification (i.e., conditional and unconditional rewriting).
- presburger is a decision procedure for Presburger arithmetic [33].
- metis is an ordered paramodulation prover.

The following tactics are also noteworthy:

- linarith is for linear arithmetic.
- meson implements Loveland's model elimination procedure.
- order is a decision procedure for orderings [108].
- smt uses external SMT solvers.

It is good to have a rough idea of the available tactics and what they can and cannot do but, in practice, we don't necessarily have to choose and write them down ourselves.

For day-to-day formalization, Isabelle/HOL offers two push-button-automation solutions: the try0 command and Sledgehammer.

The try0 command tries all of the above tactics and some more with a timeout (usually one or two seconds) in parallel, reports the successful ones to the user, and suggests to use the fastest one. When a formula seems easy, the first thing to do is to use try0 to see if a tactic can solve it.

Sledgehammer is probably Isabelle/HOL's most famous tool for proof automation [89]. It integrates external first- and higher-order automated theorem provers, including CVC4 [9], E [101, 102, 117], SPASS [34], Vampire [16, 70], veriT [24], Z3 [83], and Zipperposition [14, 36]. When invoked, Sledgehammer goes over the possibly tens of thousands of lemmas coming from the loaded theories and heuristically

selects those that could be useful to prove the formula. It then translates the goal and selected lemmas to the input formats of the external provers and invokes them in parallel. When an external prover finds and outputs a proof, Sledgehammer then tries to automatically construct a corresponding Isar proof. We can then click on the Isar proof to insert it in the source code file of the theory we are developing, after which we do not depend on the external prover. The short one-line proof calling the tactic metis in Section 2.3 was found by Sledgehammer.

But even though try0 and Sledgehammer are very useful tools, they have some limitations. First, they provide an all-or-nothing solution: either they find a proof, in which case we are done, or they do not find a proof, in which case we are not closer as before to find a proof. Second, the automatically generated proofs tend to be very compact and hard to read; it is often worthwhile to manually rewrite the proof to convey some information to human readers as to why the theorem hold.

## 2.5 Module System

Locales are a module system to manage hierarchies of parametric theories [8]. They are based on the concept of proof contexts. A formula of the form

$$\bigwedge x_1\ x_2\ \ldots\ x_n.\ P_1 \Longrightarrow P_2 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow C$$

has universally quantified variables $x_1, x_2, \ldots, x_n$, premises $P_1, P_2, \ldots, P_m$, and a conclusion $C$. Taken together, the sets of universally quantified variables and premises is called the proof context. Locales enable the user to define a named proof context and reuse it by name multiple times, thus avoiding having to repeat its components every time. The universally quantified variables are usually called *locale parameters* and the premises are usually called *locale assumptions*.

Consider the following formalization of monoids.

```
locale monoid =
  fixes f :: ‹'a ⇒ 'a ⇒ 'a› (infix "·") and e :: 'a
  assumes
    ‹⋀x y z. x · (y · z) = (x · y) · z› and
    ‹⋀x. e · x = x› and
    ‹⋀x. x · e = x›

context monoid begin
  primrec pow :: ‹nat ⇒ 'a ⇒ 'a› where
    ‹⋀x. pow 0 x = e› |
    ‹⋀n x. pow (Suc n) x = x · pow n x›

  lemma pow_add:
    fixes m :: nat and n :: nat and x :: 'a
    shows ‹pow (m + n) x = pow m x · pow n x›
  proof
    ...
  qed
end
```

The locale is named monoid and consists of a sequence of parameters, introduced by the keyword **fixes**, and a sequence of assumptions, introduced by the keyword **assumes**. The keyword **infix** specifies some special infix syntax for the parameter $f$. When working in a locale context, introduced by the command **context**, new definitions (e.g., the primitively recursive function pow) and theorems (e.g., the lemma pow_add) can be derived from the locale parameters, the locale assumptions, and previously derived terms and theorems.

Isabelle automatically defines a *locale predicate* monoid :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow$ *bool* that identify *locale interpretations* (i.e., parameters for which the assumptions hold). We can see that locale contexts really are just syntactic sugar for proof contexts by inspecting the lemma monoid.pow_add from outside the locale context.

$$\bigwedge f\ e\ m\ n\ x.\ \mathsf{monoid}\ f\ e \Longrightarrow \mathsf{monoid.pow}\ f\ e\ (m+n)\ x = f\ (\mathsf{monoid.pow}\ f\ e\ m\ x)\ (\mathsf{monoid.pow}\ f\ e\ n\ x)$$

A locale can be interpreted with the command **interpretation** by providing values for the parameters and proving that the assumptions hold.

```
interpretation add_nat: monoid ‹(+) :: nat ⇒ nat ⇒ nat› ‹0 :: nat›
proof
  show ‹⋀x y z. x + (y + z) = (x + y) + z›
    by …
next
  show ‹⋀x. 0 + x = x›
    by …
next
  show ‹⋀x. x + 0 = x›
    by …
qed
```

During interpretation, all definitions and theorems get specialized for the provided arguments and made available in the namespace add_nat. For example, the lemma monoid.pow_add gets specialized to add_nat.pow_add:

$$\bigwedge m\ n\ x.\ \mathsf{add\_nat.pow}\ (m+n)\ x = f\ (\mathsf{add\_nat.pow}\ m\ x)\ (\mathsf{add\_nat.pow}\ n\ x)$$

A new locale can also be defined by extending existing locales with more parameters and assumptions.

```
locale monoid_homomorphisms =
  mf: monoid f ef + mg: monoid g eg
  for
    f :: ‹'a ⇒ 'a ⇒ 'a› (infix "·") and ef :: 'a and
    g :: ‹'b ⇒ 'b ⇒ 'b› (infix "◇") and eg :: 'b +
  fixes map :: ‹'a ⇒ 'b›
  assumes
    ‹⋀x y. map (x · y) = (map x) ◇ (map y)› and
    ‹map ef = eg›
```

The locale monoid_homomorphisms extends two instances of the locale monoid, fixes a projection function *map* between their underlying types, and states two assumptions on the interaction of *map* with the two monoid structures. The extended locale contexts are named, so that their definitions and lemmas can be distinguished (e.g., by writing $\mathsf{m_f}$.power_add and $\mathsf{m_g}$.power_add). The keyword **for** introduces the parameters of the extended locales.

## 2.6 Notation in the Rest of This Thesis

In the rest of this thesis, we deviate from Isabelle's notation in a number of ways:

- We omit the single guillemets everywhere (e.g., we write $P\ x$ instead of $‹P\ x›$).
- We use the standard object-level syntax even for metalogical operations (i.e., we write $\forall$ instead of $\bigwedge$, $\longrightarrow$ instead of $\Longrightarrow$, and $=$ instead of $\equiv$).
- We use standard names for numeric types (e.g, we write $\mathbb{N}$ instead of *nat*).
- We prefer Boolean equivalence to Boolean equality (e.g., we write $x \longleftrightarrow y$ instead of $(x :: bool) = y$).
- We use the symbols for infix notations in lieu of the actual constant names (e.g., we write **fixes** $\prec :: 'a \Rightarrow 'a \Rightarrow bool$ instead of **fixes** less $:: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** "$\prec$")).
- We use standard numeric notation for natural numbers (i.e., we write $k$ instead of $\mathsf{Suc}^k\ 0$ and $k + n$ instead of $\mathsf{Suc}^k\ n$).
- We use a more standard syntax for set comprehension (e.g., we write $\{x \mid \exists y.\ R\ x\ y\}$ instead of $\{x \mid y.\ R\ x\ y\}$).
- We use standard set syntax for finite sets and multisets (e.g., we write $x \in \mathcal{X}$ instead of $x \in_\# \mathcal{X}$).
- We sometimes use a different name or syntax for an existing constant (e.g., we write set instead of set_mset and $|xs|$ instead of length $xs$).
- We use † for the None constructor of the *option* type but have no explicit syntax for the Some constructor (e.g., we write $x$ instead of Some $x$).
- We use $\epsilon$ for the Nil constructor of the *list* type, an infix comma for both the Cons constructor and the append function, and let lists grow from left to right (e.g., we write $xs, x$ instead of Cons $x\ xs$ and $ys, xs$ instead of append $xs\ ys$).
- We have no explicit syntax for the singleton list (e.g., we write $x$ instead of Cons $x$ Nil).
- We generally do not write the proofs but we occasionally write proof sketches. We refer the interested reader to the Isabelle/HOL formalizations for the proofs.

# Part I

# Correctness of Logical Calculi

# Chapter 3

# Preliminaries on First-Order Logic

This chapter is based on the preliminaries of two conference papers: one coauthored with Martin Bromberger and Christoph Weidenbach [25], and the other coauthored with Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret [48].

In the following chapters, we will mostly consider a first-order logic without equality. We briefly cover the most important notions relating to first-order terms (Section 3.1), formulas and clauses (Section 3.2), substitutions (Section 3.3), semantics (Section 3.4), and orderings (Section 3.5). We refer the interested reader to the literature (e.g., *Term Rewriting and All That* by Baader and Nipkow [2]) for a more in-depth presentation.

## 3.1 First-Order Terms

Given a set of variable symbols $\mathcal{V}$ and a set of function symbols $\Sigma$, a first-order *term* is defined inductively as either a variable $x \in \mathcal{V}$ or a function application $\mathtt{f}(t_1, \ldots, t_n)$ for a function symbol $\mathtt{f} \in \Sigma$ and a (possibly empty) list of terms $t_1, \ldots, t_n$ where $n$ is called an arity. A term is *ground*, or *closed*, if it does not contain variables, otherwise it is *nonground*.

A *term context* is a term with one designated position that is to be filled by another term—in other words, a term with a hole. We use the syntax $\kappa[t]$ to represent the term consisting of a subterm $t$ in a context $\kappa$. We write $\Box$ for the empty context.

In Isabelle/HOL, we represent terms with the type $('f, 'v)$ *term* and term contexts with the type $('f, 'v)$ *ctxt*, where $'f$ is an abstract type of function symbols (i.e., the set $\Sigma$) and $'v$ is an abstract type of variable symbols (i.e., the set $\mathcal{V}$). Both types are from the session `First_Order_Terms` of the *Archive of Formal Proofs* [107].

## 3.2 Formulas and Clauses

An *atomic formula*, or simply an *atom*, is a predicate symbol applied to a list of term arguments. A first-order *general formula* $\phi$ or $\psi$ is defined inductively as either $\bot$ (falsum), $\top$ (verum), an atomic formula, $\neg\phi$ (negation), $\phi \vee \psi$ (disjunction),

$\phi \wedge \psi$ (conjunction), $\phi \longrightarrow \psi$ (implication), $\phi \longleftrightarrow \psi$ (equivalence), $\forall x.\ \phi$ (universal quantification), or $\exists x.\ \phi$ (existential quantification).

We do not consider general first-order formulas directly but a simpler form instead. Firstly, we notice that logical connectives can be expressed in terms of others (e.g., $\phi \longrightarrow \psi$ can be expressed as $\neg\phi \vee \psi$). It thus suffices to consider a subset of connectives only and use this subset to define the other connectives: we chose to consider the set $\{\neg, \vee, \wedge\}$. Secondly, we notice that existential quantification can be eliminated through a process called Skolemization (e.g., $\forall x.\ \exists y.\ \mathsf{P}(x, y)$ can be Skolemized to $\forall x.\ \mathsf{P}(x, \mathsf{sk_y}(x))$ where $\mathsf{sk_y}$ is a Skolem function). It thus suffices to consider universal quantification only. In summary, we only need to consider falsum, verum, atomic formulas, negation, disjunction, and universal quantification in our simpler formula form.

A *literal* is either a positive atom $A$ or a negative atom $\neg A$. For literals, we write $L$ or $K$. The atom of a literal may be extracted with the function $\mathsf{atom}$. A literal's *polarity* identifies whether it is positive or negative. The complement of a literal, denoted by the function $\mathsf{comp}$, is itself a literal on the same atom but with opposing polarity (i.e., $\forall L.\ \mathsf{atom}\,(\mathsf{comp}\,L) = \mathsf{atom}\,L$ and $\forall L.\ (\mathsf{comp}\,L \text{ is positive}) \longleftrightarrow (L \text{ is negative})$).

A *finite multiset* is a generalization of a finite set that allows elements to occur multiple times. We say of an element with more than one occurrence that it has duplicates. For a given multiset, the multiplicity of an element, denoted by the function $\mathsf{count}$, is the number of occurrences of the element. An element is a member of a multiset, denoted by $\in$, if it has a nonzero multiplicity (e.g., $\forall x\ \mathcal{X}.\ x \in \mathcal{X} \longleftrightarrow \mathsf{count}\ \mathcal{X}\ x \neq 0$). The sum of two multisets, denoted synonymously by $+$ or $\cup$, is a multiset that adds the multiplicities of each element (i.e., $\forall \mathcal{X}\ \mathcal{Y}\ x.\ \mathsf{count}\ (\mathcal{X} + \mathcal{Y})\ x = \mathsf{count}\ \mathcal{X}\ x + \mathsf{count}\ \mathcal{Y}\ x$). The difference of two multisets, denoted by $-$, is a multiset that subtracts the multiplicities of each element (i.e., $\forall \mathcal{X}\ \mathcal{Y}\ x.\ \mathsf{count}\ (\mathcal{X} - \mathcal{Y})\ x = \mathsf{max}\ 0\ (\mathsf{count}\ \mathcal{X}\ x - \mathsf{count}\ \mathcal{Y}\ x)$).

A first-order *clausal formula*, or simply a *clause*, is a finite multiset of literals that we interpret as the disjunction of its elements. All variables in a clause are to be understood as implicitly universally quantified in that clause. For clauses we write $C$, $D$, or $E$. We use the syntax $L \vee C$ and $C \vee D$ synonymously with sums $\{L\} + C$ and $C + D$ respectively. We use the syntax $\bot$ (falsum) synonymously with the empty multiset $\{\}$.

The process of converting a general formula to a set of clausal formulas is called *clausification*. We interpret this set of clauses as the conjunction of its elements.

In Isabelle/HOL, we represent literals with the type *'a literal* and clauses with the type *'a clause* (which is an alias for the type *'a literal multiset*), where *'a* is an abstract type of atoms. Both types are from the session `Ordered_Resolution_Prover` of the *Archive of Formal Proofs* [99], which provides a reusable theory of clausal logic.

## 3.3 Substitutions

A *substitution* is a total unary function that lets us replace variables with terms. The *identity* substitution, denoted by the constant $\mathsf{id_{subst}}$, does not alter any variable. The *composition* of two substitutions, denoted by the infix operator $\circledast$, is itself a substitution that applies the two original substitutions in order from left to right. A

substitution $\sigma$ is *idempotent* if $\sigma \circledast \sigma = \sigma$.

**Remark 3.1.** *The set of substitutions forms a monoid w.r.t.* $\circledast$ *and* $\mathsf{id}_{\mathsf{subst}}$*:*

- $\forall \sigma_1\ \sigma_2\ \sigma_3.\ (\sigma_1 \circledast \sigma_2) \circledast \sigma_3 = \sigma_1 \circledast (\sigma_2 \circledast \sigma_3)$
- $\forall \sigma.\ \mathsf{id}_{\mathsf{subst}} \circledast \sigma = \sigma$
- $\forall \sigma.\ \sigma \circledast \mathsf{id}_{\mathsf{subst}} = \sigma$

A substitution $\sigma$ can be applied to a syntactic entity $X$ (e.g., a term or a clause) by writing $X\sigma$; the result is of the same type as $X$. Substitution application is left-associative (i.e., $\forall X\ \sigma_1\ \sigma_2.\ X\sigma_1\sigma_2 = (X\sigma_1)\sigma_2$).

**Remark 3.2.** *The monoid of substitutions w.r.t.* $\circledast$ *and* $\mathsf{id}_{\mathsf{subst}}$ *forms right monoid actions on terms, atoms, literals, and clauses w.r.t. substitution application. We show here the corresponding equations for clauses (the equations for terms, atoms, and literals are analogous):*

- $\forall C\ \sigma_1\ \sigma_2.\ C(\sigma_1 \circledast \sigma_2) = C\sigma_1\sigma_2$
- $\forall C.\ C\mathsf{id}_{\mathsf{subst}} = C$

The *domain* of a substitution, denoted by the function $\mathsf{dom}$, is the set of all variables altered when applying the substitution (i.e., $\forall \sigma.\ \mathsf{dom}\ \sigma = \{x \mid x\sigma \neq x\}$). The *restriction* of a substitution to a variable set, denoted by the infix operator $\restriction$, is itself a substitution that restricts the original substitution's domain to the given variable set (i.e., $\forall \sigma\ \mathcal{X}.\ (\forall x \in \mathcal{X}.\ x(\sigma\restriction_{\mathcal{X}}) = x\sigma) \wedge (\forall x \notin \mathcal{X}.\ x(\sigma\restriction_{\mathcal{X}}) = x)$).

A substitution $\rho$ is a *renaming substitution*, or simply a *renaming*, if it is injective and only maps variables to variables. Each renaming $\rho$ has an *inverse renaming* $\rho^{-1}$, which is itself a substitution that cancels $\rho$ out.

**Remark 3.3.** *The submonoid of renaming substitutions w.r.t.* $\circledast$ *and* $\mathsf{id}_{\mathsf{subst}}$ *forms a right-group w.r.t. the inverse renaming operation:*

- $\forall \rho.\ \rho$ *is a renaming* $\longrightarrow \rho \circledast \rho^{-1} = \mathsf{id}_{\mathsf{subst}}$

A substitution $\gamma$ is a *grounding substitution*, or simply a *grounding* for a syntactic entity $X$ if $X\gamma$ is ground.

A substitution $\upsilon$ is a *unifier* for a term set $\mathcal{T}$ if its application makes all elements of the set equal (i.e., if $\forall t_1 \in \mathcal{T}.\ \forall t_2 \in \mathcal{T}.\ t_1\upsilon = t_2\upsilon$). A substitution $\mu$ is a *most general unifier* (MGU) for a term set $\mathcal{T}$ if $\mu$ is a unifier for $\mathcal{T}$ and if any other unifier is equal to $\mu$ composed with some appropriate substitution (i.e., if $\forall \upsilon.\ (\upsilon$ is a unifier for $\mathcal{T}) \longrightarrow \exists \sigma.\ \mu \circledast \sigma = \upsilon$). A substitution $\mu$ is an *idempotent, most general unifier* (IMGU) for a term set $\mathcal{T}$ if $\mu$ is a unifier for $\mathcal{T}$ and if $\mu$ is a left identity for any other unifier (i.e., if $\forall \upsilon.\ (\upsilon$ is a unifier for $\mathcal{T}) \longrightarrow \mu \circledast \upsilon = \upsilon$). Equivalently, $\mu$ is an IMGU for term set $\mathcal{T}$ if, and only if, $\mu$ is both idempotent and a MGU for $\mathcal{T}$.

When formalizing logical calculi, IMGUs are preferable because they allow to apply a unifier to a term both directly and after applying an IMGU (i.e., $\forall \mathcal{T}\ \mu\ \upsilon.\ (\mu$ is an IMGU
for $\mathcal{T}) \longrightarrow (\upsilon$ is a unifier for $\mathcal{T}) \longrightarrow (\forall t \in \mathcal{T}.\ t\mu\upsilon = t(\mu \circledast \upsilon) = t\upsilon)$). Nonidempotent MGU do not have this property as the following counter-example shows.

**Example 3.4.** *Let* $\mathsf{f}$ *be a function symbol. Let* $w$*,* $x$*,* $y$*, and* $z$ *be variable symbols. Let* $\mathsf{a}$*,* $\mathsf{b}$*, and* $\mathsf{c}$ *be ground terms. Consider the (nonground) terms* $\mathsf{t}_1 = \mathsf{f}(x, y, z)$ *and* $\mathsf{t}_2 = \mathsf{f}(w, y, z)$*, the grounding substitution* $\gamma = \{x \mapsto \mathsf{a}, y \mapsto \mathsf{b}, z \mapsto \mathsf{c}, w \mapsto \mathsf{a}\}$ *that unifies* $\mathsf{t}_1$ *and* $\mathsf{t}_2$*, and the nonidempotent MGU* $\mu = \{x \mapsto w, y \mapsto z, z \mapsto y\}$ *for* $\mathsf{t}_1$ *and* $\mathsf{t}_2$*. Observe that* $\mathsf{t}_1\gamma = \mathsf{t}_2\gamma = \mathsf{f}(\mathsf{a}, \mathsf{b}, \mathsf{c}) \neq \mathsf{f}(\mathsf{a}, \mathsf{c}, \mathsf{b}) = \mathsf{t}_2\mu\gamma = \mathsf{t}_1\mu\gamma$*.*

In published literature, authors often claim to be using an MGU when they actually need an IMGU; the idempotency requirement is kept implicit because standard algorithms for computing MGUs actually produce IMGUs.

The result of applying a substitution $\sigma$ to a syntactic entity $X$ (i.e., $X\sigma$) is called an *instance* of $X$. The *instances* of a syntactic entity $X$ are the results of applying all possible substitutions to $X$ (i.e., $\{X' \mid \exists\sigma.\ X\sigma = X'\}$). The *ground instances* of a syntactic entity $X$ are the results of applying all possible grounding substitutions to $X$. For clauses, this is denoted by the function $\mathsf{gnd}_{\mathsf{cls}}$ (i.e., $\forall C.\ \mathsf{gnd}_{\mathsf{cls}}\ C = \{C\gamma \mid C\gamma \text{ is ground}\}$). We lift $\mathsf{gnd}_{\mathsf{cls}}$ to clause sets, denoted by $\mathsf{gnd}_{\mathsf{clss}}$, by taking the union of the ground instances of all the clauses in the set (i.e., $\forall N.\ \mathsf{gnd}_{\mathsf{clss}}\ N = (\bigcup C \in N.\ \mathsf{gnd}_{\mathsf{cls}}\ C)$). Note that $\mathsf{gnd}_{\mathsf{cls}}$ and $\mathsf{gnd}_{\mathsf{clss}}$ have no effect if the clauses are already ground (i.e., $\forall C.\ (C \text{ is ground}) \longrightarrow \mathsf{gnd}_{\mathsf{cls}}\ C = \{C\}$ and $\forall N.\ (N \text{ is ground}) \longrightarrow \mathsf{gnd}_{\mathsf{clss}}\ N = N$) and that $\mathsf{gnd}_{\mathsf{clss}}$ is idempotent (i.e., $\forall N.\ \mathsf{gnd}_{\mathsf{clss}}(\mathsf{gnd}_{\mathsf{clss}}\ N) = \mathsf{gnd}_{\mathsf{clss}}\ N$).

In Isabelle/HOL, we developed a reusable theory of abstract substitution; it is available in the session `Abstract_Substitution` of the *Archive of Formal Proofs* [41]. The theory is based on the concept of monoid actions where both the object type (e.g., terms, atoms, literals, or clauses) and the substitution type (e.g., a function from variables to terms or a list of variable-term pairs) are kept abstract so future formalizations can use the concrete types that best fit their need. Many of the above definitions as well as useful lemmas are expressed entirely in terms of monoid actions and automatically provided.

## 3.4 Semantics

We use a Herbrand interpretation to define the semantics of a formula. An *Herbrand interpretation* is a set of ground atoms that are considered true. The truth value of a ground literal w.r.t. an interpretation, denoted by the infix relation $\models_{\mathsf{lit}}$, depends on the literals polarity: a positive literal is considered true if its atom is in the interpretation and a negative atom is considered true if its atom is not in the interpretation (i.e., $\forall \mathcal{I}\ L.\ \mathcal{I} \models_{\mathsf{lit}} L \longleftrightarrow (L \text{ is positive} \longleftrightarrow \mathsf{atom}\ L \in \mathcal{I})$). A ground clause is considered true w.r.t. an interpretation, denoted by $\models_{\mathsf{cls}}$, if at least one of its literals is true (i.e., $\forall \mathcal{I}\ C.\ \mathcal{I} \models_{\mathsf{cls}} C \longleftrightarrow (\exists L \in C.\ \mathcal{I} \models_{\mathsf{lit}} L)$). A ground clause set is considered true w.r.t. an interpretation, denoted by $\models_{\mathsf{clss}}$, if all its clauses are true (i.e., $\forall \mathcal{I}\ N.\ \mathcal{I} \models_{\mathsf{clss}} N \longleftrightarrow (\forall C \in N.\ \mathcal{I} \models_{\mathsf{cls}} C)$).

An interpretation $\mathcal{I}$ is called a *model* of a clause $C$ or clause set $N$ if $\mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}_{\mathsf{cls}}\ C)$ or $\mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}_{\mathsf{clss}}\ N)$ respectively. A clause or a clause set is *satisfiable* if it has a model; otherwise, it is unsatisfiable. A clause set *logically implies*, or *entails*, another clause set, denoted by $\models$, if any model of the former is also a model of the later (i.e., $\forall N_1\ N_2.\ N_1 \models N_2 \longleftrightarrow (\forall \mathcal{I}.\ \mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}_{\mathsf{clss}}\ N_1) \longrightarrow \mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}_{\mathsf{clss}}\ N_2))$).

In Isabelle/HOL, most of these definitions are from the session `Ordered_Resolution_Prover` of the *Archive of Formal Proofs* [99], which provides a reusable theory of Herbrand interpretations.

## 3.5 Orderings

A binary relation $\prec$ is a (strict) *partial ordering*, or simply an *ordering*, if it is transitive (i.e., $\forall x\, y\, z.\ x \prec y \longrightarrow y \prec z \longrightarrow x \prec z$) and asymmetric (i.e., $\forall x\, y.\ x \prec y \longrightarrow y \not\prec x$). Note that a partial ordering is also irreflexive (i.e., $\forall x.\ x \not\prec x$) because any transitive relation is irreflexive if and only if it is asymmetric. An ordering $\prec$ is a (strict) *total ordering*, a.k.a. a *linear ordering*, if it is total (i.e., $\forall x\ y.\ x \prec y \vee x = y \vee y \prec x$). An ordering $\prec$ is a (strict) *well-founded ordering*, if it is well-founded (i.e., there is no infinite descending chain). Sometimes, a binary relation has a property only on a subset of the universe of its elements. A binary relation $\prec$ is a (strict) *ordering* on a set (or multiset) $\mathcal{X}$ if the corresponding properties (e.g., transitivity and asymmetry) hold for all elements of $\mathcal{X}$.

**Example 3.5.** *The Knuth–Bendix ordering [69] is well-founded on all terms and total on ground terms.*

An ordering $\prec_{\mathsf{atm}}$ on atoms can be lifted to an ordering $\prec_{\mathsf{lit}}$ on literals by first comparing the atoms w.r.t. $\prec_{\mathsf{atm}}$ and, if the atoms are the same, specifying one polarity as less than the other. We chose to consider positive literals to be less than negative literals (e.g., $\forall A.\ A \prec_{\mathsf{lit}} \neg A$). An ordering $\prec_{\mathsf{lit}}$ on literals can be lifted to an ordering $\prec_{\mathsf{cls}}$ on clauses by taking a multiset extension of $\prec_{\mathsf{lit}}$. There are several equivalent alternatives for this extension (e.g., the Dershowitz–Manna extension [37] or the Huet–Oppen extension [66]).

**Example 3.6.** *Let $\prec$ be an ordering. The Huet–Oppen multiset extension $\prec_{HO}$ lifts the ordering $\prec$ to an ordering on multisets.*

$$\forall \mathcal{X}\ \mathcal{Y}.\ \mathcal{X} \prec_{HO} \mathcal{Y} \longleftrightarrow$$
$$\mathcal{X} \neq \mathcal{Y} \wedge (\forall x.\ \mathsf{count}\ \mathcal{Y}\ x < \mathsf{count}\ \mathcal{X}\ x \longrightarrow (\exists y.\ x \prec y \wedge \mathsf{count}\ \mathcal{X}\ y < \mathsf{count}\ \mathcal{Y}\ y))$$

An element $x$ is *minimal* in a finite multiset $\mathcal{X}$ w.r.t. a strict partial ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and no element of $\mathcal{X}$ is less than $x$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ y \not\prec x)$). Analogously, an element $x$ is *maximal* in a finite multiset $\mathcal{X}$ w.r.t. a strict partial ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and no element of $\mathcal{X}$ is greater than $x$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ x \not\prec y)$).

An element $x$ is *strictly minimal* in a finite multiset $\mathcal{X}$ w.r.t. a strict partial ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and no other element of $\mathcal{X}$ is less than or equal to $x$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X} - \{x\}.\ y \not\preceq x)$). Analogously, an element $x$ is *strictly maximal* in a finite multiset $\mathcal{X}$ w.r.t. a strict partial ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and no other element of $\mathcal{X}$ is greater than or equal to $x$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X} - \{x\}.\ x \not\preceq y)$).

An element $x$ is the *least* in a finite multiset $\mathcal{X}$ w.r.t. a strict total ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and it is less than all other elements of $\mathcal{X}$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X} - \{x\}.\ x \prec y)$). Analogously, an element $x$ the *greatest* in a finite multiset $\mathcal{X}$ w.r.t. a strict total ordering $\prec$ on $\mathcal{X}$ if $x$ is in $\mathcal{X}$ and it is greater than all other elements of $\mathcal{X}$ (i.e., if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ y \prec x)$).

In these six definitions, the relation $\prec$ only has to fulfill the ordering requirements on the elements of $\mathcal{X}$ and is free to be defined in any other way for elements not in $\mathcal{X}$. We sometimes omit to specify the ordering when it is clear from the context.

The two notions of maximal and strictly maximal elements coincide except for their handling of duplicates: A maximal element can have duplicates, whereas a

strictly maximal element cannot. If the ordering is not total, a multiset can have multiple maximal or strictly maximal elements. The same observations apply to the notions of minimal and strictly minimal elements.

**Example 3.7.** *Consider the multiset* $\mathcal{X} = \{🍎, 🍊, 🍐, 🍐\}$ *containing an apple, an orange, and two pears. Consider a partial fruit ordering* $\prec$ *where apples, oranges, and pears are all mutually incomparable. The multiset* $\mathcal{X}$ *has three elements maximal w.r.t.* $\prec$*:*

- 🍎 *is maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍎 \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ 🍎 \not\prec y)$ *holds.*
- 🍊 *is maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍊 \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ 🍊 \not\prec y)$ *holds.*
- 🍐 *is maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍐 \in \mathcal{X} \wedge (\forall y \in \mathcal{X}.\ 🍐 \not\prec y)$ *holds.*

*However, the multiset* $\mathcal{X}$ *has only two elements strictly maximal w.r.t.* $\prec$*:*

- 🍎 *is strictly maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍎 \in \mathcal{X} \wedge (\forall y \in \{🍊, 🍐, 🍐\}.\ 🍎 \not\preceq y)$ *holds.*
- 🍊 *is strictly maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍊 \in \mathcal{X} \wedge (\forall y \in \{🍎, 🍐, 🍐\}.\ 🍊 \not\preceq y)$ *holds.*

*Note that* 🍐 *is not strictly maximal in* $\mathcal{X}$ *w.r.t.* $\prec$ *because* $🍐 \in \mathcal{X} \wedge (\forall y \in \{🍎, 🍊, 🍐\}.\ 🍐 \not\preceq y)$ *does not hold (because* $🍐 \not\preceq 🍐$ *does not hold).*

The notion of greatest element is based on a total ordering and provides a strong unicity guarantee (i.e., $\forall \mathcal{R}\ \mathcal{X}\ x\ x'.$ ($x$ is the greatest element in $\mathcal{X}$ w.r.t. $\mathcal{R}$) $\longrightarrow$ ($x'$ is the
greatest element in $\mathcal{X}$ w.r.t. $\mathcal{R}$) $\longrightarrow x = x'$). This is the reason why we refer to "the" greatest element while we refer to "a" (strictly) maximal element. The same observation applies to the notion of least element.

The notions of (strictly) minimal, (strictly) maximal, least, and greatest element generalize to sets and finite sets. Because sets have no duplicates, the distinction between strictly and nonstrictly minimal or maximal vanishes.

In Isabelle/HOL, we developed a reusable theory of minimal, maximal, least and greatest elements in sets, finite sets, and finite multisets; it is available in the session `Min_Max_Least_Greatest` of the *Archive of Formal Proofs* [42]. The theory keeps the ordering assumptions to a minimum so that it can be used, e.g, with nontotal orderings such as the Knuth–Bendix ordering. This new theory was necessary because the Isabelle/HOL distribution, to the best of our knowledge, only contains a theory of least and greatest elements w.r.t. a total ordering, which we cannot use when we have only a partial ordering.

# Chapter 4

# Ground Superposition

This chapter is based on a conference paper coauthored with Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret [48]. Section 4.4 describes my formalization work. Section 4.5 summarizes Balazs Toth's formalization work. To avoid any potential conflict between this thesis and Balazs' future thesis, I replaced the content of Section 4.5 with a short summary of the main results. The interested reader can refer to the paper for more details.

## 4.1 Introduction

Superposition is a highly successful proof calculus for reasoning about first-order logic with equality designed by Bachmair and Ganzinger [4, 5]. It is implemented in many automatic theorem provers, including Drodi, E [101, 102], iProver [50], SPASS [34], Vampire [70], and Zipperposition [36]. In addition, higher-order variants of the calculus are implemented in Duper, E [117], Leo-III [105], Vampire [16], and Zipperposition [14], and an arithmetic-capable variant is implemented in Beagle [10].

Superposition provers work by refutation and saturation. They operate on a clause set, which initially consists of the clausified input problem in which the conjecture appears negated. Inferences are performed using clauses from this set as premises; the conclusions of inferences are added to the set. The prover stops when the empty clause $\bot$, denoting falsehood, is derived or when no more inferences are possible.

Consider the problem of proving $\mathtt{f}(\mathtt{b}) \approx \mathtt{f}(\mathtt{a})$ from $\mathtt{b} \approx \mathtt{a}$, where $\approx$ denotes equality. After negating the conjecture, we obtain the clause set $\{\mathtt{b} \approx \mathtt{a}, \mathtt{f}(\mathtt{b}) \not\approx \mathtt{f}(\mathtt{a})\}$. The superposition calculus includes an inference rule called superposition that uses the first clause to rewrite the second clause to $\mathtt{f}(\mathtt{a}) \not\approx \mathtt{f}(\mathtt{a})$. This new clause is added to the clause set. At this point, a unary inference rule called equality resolution uses $\mathtt{f}(\mathtt{a}) \not\approx \mathtt{f}(\mathtt{a})$ to derive $\bot$.

During the saturation, the prover can delete clauses considered redundant, and it does not need to perform inferences considered redundant. For example, if the clause set contains $\mathtt{b} \approx \mathtt{a}$, then the clauses $\mathtt{f}(\mathtt{b}) \approx \mathtt{f}(\mathtt{a})$ and $\mathtt{b} \approx \mathtt{a} \vee \mathtt{b} \not\approx \mathtt{c}$ are redundant. Deletion of redundant clauses helps reduce the clause explosion caused by saturation.

The inference rules of the superposition calculus are *sound*, meaning that the conclusion of each rule is entailed by the premises. This is easy to prove. What is much harder to show is that the calculus is *refutationally complete*: If a clause set is

unsatisfiable and saturated (up to redundancy), then it contains $\bot$. We care about completeness because a complete calculus is likely to yield a higher success rate in practice than an incomplete one. Moreover, the completeness proof serves as a guide during the development of the calculus: Only inferences that are needed in the proof must be performed.

When developing proof calculi for first-order logic and beyond, it often helps to first develop a calculus that works on ground (i.e., variable-free) clauses. We can then lift it systematically to the nonground level. This approach cleanly separates concerns. It is common in the literature [13, 14, 15, 16, 17, 94] and is supported by the *saturation framework* developed by Bachmair [3, Section 4] and extended by Waldmann et al. [119], a collection of pen-and-paper results useful to establish the refutational completeness of saturation calculi and provers.

For superposition, Bachmair and Ganzinger's completeness proof [5] does not separate the ground and nonground aspects. Waldmann et al. give some hints on how to instantiate the framework to obtain a modular proof that separates these aspects—see their Examples 3, 4, 28, 34, 46, and 54. Our main contributions are twofold. First, we elaborated these hints into a 15-page proof text [118] (summarized here in Section 4.3). Second, following this detailed *blueprint*, we formalized in Isabelle/HOL [85] the refutational completeness of ground superposition (Section 4.4) and lifted it to derive the refutational completeness of the nonground calculus (Section 4.5). We also proved soundness.

The separation of concerns, apart from allowing different people to work independently on different parts of the formalization, simplifies the completeness proof. On the ground level, there is no need to rename variables apart or to perform unification. On the nonground level, an inference overapproximates a set of ground inferences. Intuitively, this means that every inference on ground clauses can be simulated by inferences on corresponding nonground clauses. For superposition inferences, this roughly means that if $D\gamma_1$ and $E\gamma_2$ are premises of a nonredundant ground inference yielding $C$, where $\gamma_1, \gamma_2$ are substitutions, then there exists an inference with $D$ and $E$ as premises and whose conclusion is a generalization of $C$.

A difficulty arises on the nonground level because the calculus is optimized to avoid superposition *into* variables. For example, given the clause set $\{\mathtt{b} \approx \mathtt{a},\ \mathtt{f}(x) \not\approx \mathtt{c}\}$, a superposition inference unifying $\mathtt{b}$ with $x$ would yield the conclusion $\mathtt{f}(\mathtt{a}) \not\approx \mathtt{c}$, but the calculus excludes this inference. Intuitively, since $\mathtt{f}(\mathtt{a}) \not\approx \mathtt{c}$ is an instance of $\mathtt{f}(x) \not\approx \mathtt{c}$, we would expect the inference to be unnecessary, but this must be justified in general.

The Isabelle formalization relies on the first-order terms and related notions from the `IsaFoR` library [110]. It also uses the Isabelle version of the saturation framework [113]. The formalization validates the pen-and-paper proof: We found only one easy-to-repair mistake and one unnecessary assumption. The formalization can serve as a reference for refutational completeness of superposition, an important result in automated reasoning. It could also serve as the basis of a verified executable prover.

Ours is not the first formalization of superposition in a proof assistant, or even in Isabelle/HOL. Our predecessor is Peltier, who formalized a generalization of superposition and published his result in the *Archive of Formal Proofs* (*AFP*) [90]. However, his proof is monolithic, mixing ground and nonground aspects. By using the saturation framework, we get a clearer proof structure and immediately obtain

the completeness of an abstract prover based on superposition [119, Lemma 10] as well as the completeness of various saturation procedures [119, Section 4].

Our Isabelle formalization is available in the *Archive of Formal Proofs* [47]. The underlying pen-and-paper proof is available online [118]. Our work is part of the IsaFoL (Isabelle Formalization of Logic) effort [18].

## 4.2 Background

We briefly introduce prerequisites, the superposition calculus, and the saturation framework.

**Prerequisites.** We consider an untyped first-order logic with equality. *Terms* and *term contexts* are defined in Section 3.1. An *atom* is an unordered pair of terms, typically written as an equation $t \approx t'$. *Literals* and *clauses* are defined in Section 3.2. We write $t \not\approx t'$ as an abbreviation for the negative literal $\neg(t \approx t')$. *Substitutions* are defined in Section 3.3. The *semantics* of clauses is defined in Section 3.4. *Orderings* are defined in Section 3.5. In the following, (strictly) maximal elements in a clause are w.r.t. the literal ordering $\prec_{\mathsf{lit}}$ unless we explicitly state otherwise.

We let $s$, $t$, and $u$ range over terms, $K$ and $L$ range over literals, and $C$, $D$, and $E$ range over clauses.

**The Superposition Calculus.** Bachmair and Ganzinger's superposition calculus [4, 5] belongs to a class of proof calculi for automatic provers known as saturation calculi. A saturation prover takes a set of formulas, usually clauses, as input and processes it by performing two operations: First, it derives new formulas from the old ones and adds them to the set. Second, it deletes superfluous formulas from the set. This process is repeated until the prover either finds $\bot$ or reaches a state in which it is not required to add further formulas.

Abstractly, the calculus can be defined by two components: a set of inferences

$$\frac{C_n \quad \cdots \quad C_1}{C_0}$$

indicating that the formula $C_0$ (the *conclusion*) must be added to the set whenever the formulas $C_n, \ldots, C_1$ (the *premises*) are already present, and a redundancy criterion that describes which inferences are unnecessary and which formulas may be deleted from the set.

For the superposition calculus, the inferences are given by three schematic inference rules. The first one is

$$\frac{\overbrace{t \approx t' \vee D'}^{D} \qquad \overbrace{\kappa[u] \bowtie u' \vee E'}^{E}}{\underbrace{(\kappa[t'\rho] \bowtie u' \vee E' \vee D'\rho)\mu}_{C}}\ \text{superposition}$$

where the clauses $D$ and $E$ are the premises, $C$ is the conclusion, $\bowtie$ is either $\approx$ or $\not\approx$, $u$ is a nonvariable subterm occurring in a context $\kappa$ in clause $E$, $\rho$ is an arbitrary but fixed renaming that is chosen so that $D\rho$ and $E$ are variable-disjoint, and $\mu$ is an IMGU of $t\rho$ and $u$.

The other two rules are

$$\frac{\overbrace{t \not\approx t' \vee D'}^{D}}{\underbrace{D'\mu}_{C}}\text{ equality resolution}$$

where $\mu$ is an IMGU of $t$ and $t'$, and

$$\frac{\overbrace{u \approx u' \vee t \approx t' \vee D'}^{D}}{\underbrace{(u' \not\approx t' \vee u \approx t' \vee D')\mu}_{C}}\text{ equality factoring}$$

where $\mu$ is an IMGU of $t$ and $u$.

To reduce the number of inferences that need to be computed during the saturation, the inference rules above are equipped with ordering restrictions. Let $\prec_{\mathsf{t}}$ be an ordering on terms that is stable under grounding substitutions, and whose ground restriction is well-founded, total, and compatible with contexts, and has the subterm property. The term ordering $\prec_{\mathsf{t}}$ is extended to a literal ordering and a clause ordering in the following way: To every positive literal $t \approx t'$, we assign the multiset $\{t, t'\}$, to every negative literal $t \not\approx t'$, we assign the multiset $\{t, t, t', t'\}$. The literal ordering $\prec_{\mathsf{lit}}$ compares these multisets using the multiset extension of $\prec_{\mathsf{t}}$. The clause ordering $\prec_{\mathsf{cls}}$ compares clauses by comparing their multisets of literals using the multiset extension of $\prec_{\mathsf{lit}}$.

We impose the following ordering restrictions on the inferences above: **(1)** If $L$ is the first literal in a premise $D$ or $E$, it must be maximal in that premise w.r.t. $\prec_{\mathsf{lit}}$ (after applying the substitution); **(2)** if additionally $L$ is a positive equation in a superposition inference, it must be strictly maximal; **(3)** except in equality resolution inferences, the right-hand side of the equation or negated equation $L$ may not be larger than or equal to the left-hand side w.r.t. $\prec_{\mathsf{t}}$; and **(4)** in superposition inferences, $D\rho\mu$ may not be larger than or equal to $C\mu$ w.r.t. $\prec_{\mathsf{cls}}$.

The impact of ordering restrictions is limited by the requirement that the ordering has to be stable under grounding substitutions, that is, that $t \prec_{\mathsf{t}} t'$ implies $t\gamma \prec_{\mathsf{t}} t'\gamma$ for every grounding substitution $\gamma$. Stability under grounding substitutions entails that terms such as $\mathtt{f}(x)$ and $\mathtt{f}(y)$ are necessarily incomparable. (If we had $\mathtt{f}(x) \prec_{\mathsf{t}} \mathtt{f}(y)$, we could conclude $\mathtt{f}(\mathtt{b}) \prec_{\mathsf{t}} \mathtt{f}(\mathtt{b})$, contradicting irreflexivity.) Consequently, in the worst case, all literals in a clause can be maximal. For clauses with negative literals, this effect can be remedied using a *selection function* that overrides the ordering restrictions. This is a function that maps every clause to a submultiset of its negative literals (usually a singleton set or the empty set). The ordering conditions above are then modified so that if at least one literal in a clause is selected, then the maximality conditions for literals are applied to the selected submultiset instead of the original clause. This means that only inferences that involve literals that are maximal among the selected literals need to be performed.

The local restrictions that are imposed on individual inferences are supplemented by a global redundancy criterion for clauses and inferences. Bachmair and Ganzinger's *standard redundancy criterion* is defined as follows:[1] A ground clause $C$ is redundant

[1] Bachmair and Ganzinger changed their notation after their first publication on the superposition

w.r.t. a set $N$ of ground clauses if it is entailed by clauses in $N$ that are smaller than $C$ w.r.t. $\prec_{\mathsf{cls}}$. A nonground clause $C$ is redundant w.r.t. a set $N$ of nonground clauses if every ground instance of $C$ is redundant w.r.t. the set of all ground instances of clauses in $N$. A ground inference (i.e., an inference with ground premises and ground conclusion) is redundant w.r.t. a set $N$ of ground clauses if its conclusion is entailed by clauses in $N$ that are smaller than the maximal premise. A nonground inference is redundant w.r.t. a set $N$ of nonground clauses if every ground instance of the inference is redundant w.r.t. the set of all ground instances of clauses in $N$.

Redundant clauses may be deleted from the clause set during a saturation; redundant inferences need not be computed. In particular, inferences whose conclusion is already contained in the clause set are always redundant.

**The Saturation Framework.** In their article in the *Handbook of Automated Reasoning* [3], Bachmair gave a general account of components and properties of saturation calculi such as inferences, redundancy criteria, fairness, refutational completeness, and the connections between these. The framework by Waldmann et al. [119] extended this to include a general treatment of lifting, subsumption, and prover architectures. We summarize the main results needed for the present report.

Let $F$ be a set of formulas, and $\models$ be a consequence relation on $F$. An *$F$-inference* is an inference with premises and conclusion in $F$. An *$F$-inference system* *Inf* is a set of $F$-inferences. If $N \subseteq F$, we write $\mathit{Inf}(N)$ for the set of all inferences in *Inf* with premises in $N$.

Let $\mathit{Red}_{\mathsf{I}}$ be a function from sets of formulas to sets of inferences; let $\mathit{Red}_{\mathsf{F}}$ be a function from sets of formulas to sets of formulas. The pair $\mathit{Red} = \langle \mathit{Red}_{\mathsf{I}}; \mathit{Red}_{\mathsf{F}} \rangle$ is a *redundancy criterion* for *Inf* if it satisfies the following conditions:

1. if $N \models \{\bot\}$, then $N - \mathit{Red}_{\mathsf{F}}(N) \models \{\bot\}$;
2. if $N \subseteq N'$, then $\mathit{Red}_{\mathsf{F}}(N) \subseteq \mathit{Red}_{\mathsf{F}}(N')$ and $\mathit{Red}_{\mathsf{I}}(N) \subseteq \mathit{Red}_{\mathsf{I}}(N')$;
3. if $N' \subseteq \mathit{Red}_{\mathsf{F}}(N)$, then $\mathit{Red}_{\mathsf{F}}(N) \subseteq \mathit{Red}_{\mathsf{F}}(N - N')$ and $\mathit{Red}_{\mathsf{I}}(N) \subseteq \mathit{Red}_{\mathsf{I}}(N - N')$; and
4. if the conclusion of an inference in *Inf* is in $N$, then the inference is in $\mathit{Red}_{\mathsf{I}}(N)$.

Inferences in $\mathit{Red}_{\mathsf{I}}(N)$ and formulas in $\mathit{Red}_{\mathsf{F}}(N)$ are called redundant w.r.t. $N$.

A saturation prover for a calculus $\langle \mathit{Inf}; \mathit{Red} \rangle$ gets a set of formulas $N_0 \subseteq F$ as input and generates a sequence $N_0, N_1, \ldots$ of sets of formulas by adding newly computed formulas and by deleting unnecessary formulas. We require that in every step the deleted formulas are redundant w.r.t. the remaining ones, that is, $N_{i+1} - N_i \subseteq \mathit{Red}(N_{i+1})$ for every $i$. We call the sequence $N_0, N_1, \ldots$ a *derivation*. The set $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$ of persistent formulas is called the *limit* of the derivation. The derivation is *fair* if every inference from persistent formulas eventually becomes redundant, that is, if $\mathit{Inf}(N_\infty) \subseteq \bigcup_i \mathit{Red}_{\mathsf{I}}(N_i)$. (Recall that an inference becomes redundant in particular if its conclusion is added to the set of formulas.) The calculus $\langle \mathit{Inf}; \mathit{Red} \rangle$ is *dynamically refutationally complete* if for every set $N_0$ with $N_0 \models \{\bot\}$ and every fair derivation $N_0, N_1, \ldots$, the formula $\bot$ is eventually derived, that is, $\bot \in \bigcup_i N_i$.

Proving the dynamic refutational completeness of the calculus $\langle \mathit{Inf}; \mathit{Red} \rangle$ directly is usually difficult. Fortunately, dynamic refutational completeness can be shown

calculus. What is known as redundancy nowadays was called compositeness in their papers [4, 5].

to be equivalent to another property, namely static refutational completeness: A set $N \subseteq F$ is *saturated* w.r.t. *Inf* and *Red* if $\mathit{Inf}(N) \subseteq \mathit{Red}_{\mathsf{I}}(N)$. The calculus $\langle \mathit{Inf}; \mathit{Red} \rangle$ is *statically refutationally complete* if for every saturated set $N$ we have that $N \models \bot$ implies $\bot \in N$.

To prove the static (and thus dynamic) refutational completeness of a calculus, it is usually convenient to start with a ground version of the calculus. The completeness result for the nonground calculus can then be obtained from the completeness result for the ground calculus by lifting, using a suitable *grounding* function that maps nonground formulas to sets of ground formulas and nonground inferences to sets of ground inferences. The framework also shows how to deal with redundancy criteria that are defined as intersections of other redundancy criteria (a technique that we will need to handle selection functions in the lifting process), how to integrate *subsumption* into the redundancy criterion (so that, e.g., $x \approx \mathsf{a}$ makes its instance $\mathsf{b} \approx \mathsf{a}$ redundant), and how to obtain completeness results for implementations of the calculus in various prover architectures.

The framework has been formalized in Isabelle/HOL and extended by Tourret and Blanchette [21, 111, 113]. The present work builds on this formalization.

## 4.3 Proof Outline

Static refutational completeness can be stated as follows:

**Theorem 4.1.** *For every set $N$ that is saturated w.r.t. the superposition calculus, if $N$ entails $\bot$, then $\bot \in N$.*

Equivalently: For every saturated set $N$ such that $\bot \notin N$, there exists a model of $N$. Bachmair and Ganzinger's original proof [5, Section 4] uses a monolithic approach. Our proof is more modular and proceeds in two clearly separated steps:

1. Given a ground clause set $M$ saturated w.r.t. ground inferences, we construct a model of $M$.
2. We show that if a clause set $N$ is saturated w.r.t. nonground inferences, then its grounding $N_{\mathsf{G}} = \{C\gamma \mid C \in N \wedge C\gamma \text{ is ground}\}$ is saturated w.r.t. ground inferences. Hence, by step 1, there exists a model of $N_{\mathsf{G}}$, which is also a model of $N$.

In step 1, we construct a confluent and terminating term rewriting system $R_\infty$ and use it to define an interpretation that equates all terms that share the same normal form w.r.t. $R_\infty$, and no others. For example, if $R_\infty = \{\mathsf{b} \to \mathsf{a}\}$, then the associated interpretation makes $\mathsf{f}(\mathsf{b}) \approx \mathsf{f}(\mathsf{a})$ true and $\mathsf{c} \approx \mathsf{a}$ false. The system $R_\infty$ is built incrementally. We start with $\{\}$ and traverse the clauses in $M$ from the smallest clause following the ordering $\prec_{\mathsf{cls}}$. For each clause $C \in M$, if $C$ is true in the current interpretation, there is nothing to do. Otherwise, we extend the term rewriting system with a rewrite rule that attempts to make $C$ true without affecting the truth of earlier, smaller clauses. While this process might fail in general, it will always produce a model of $M$ if $M$ is saturated.

In step 2, we must show that saturation on the nonground level implies saturation on the ground level. Via a result from the saturation framework, this amounts to showing that there exist nonground inferences corresponding to all nonredundant

ground inferences of the calculus. A subtlety is that the calculus avoids superposition inferences into variables. Thus there might exist ground inferences that are not reflected on the nonground level. However, we can show that all such ground inferences are redundant. Another concern is the selection function $S$. In general, we cannot assume that $S$ is stable under substitutions (i.e., $\forall C\,\sigma.\ (S\,C)\sigma = S(C\sigma)$), but without this assumption it is hard to relate the ground and nonground levels. The solution is provided by the saturation framework, which allows us to simultaneously lift all ground selection functions to the nonground level.

## 4.4 The Ground Proof

On the ground level, we reuse theories [80] from the `IsaFoR` project for ground terms, of type $'f\ gterm$, and ground term contexts, of type $'f\ gctxt$; the type variable $'f$ represents function symbols. Isabelle types use a postfix, space-separated notation (e.g., sets of Boolean have type $bool\ set$). Ground atoms have type $'f\ gatom$, which is a synonym for $'f\ gterm\ uprod$, i.e., unordered pair of ground terms. Ground literals have type $'f\ gatom\ literal$. Ground clauses have type $'f\ gclause$, which is a synonym for $'f\ gatom\ literal\ multiset$. Isabelle multisets are always finite.

We start the formalization by introducing a locale, or module, that fixes an ordering on ground terms and specifies some assumptions on this ordering:

**locale** ground_ordering =
  **fixes** $(\prec_\mathsf{t}) :: 'f\ gterm \Rightarrow 'f\ gterm \Rightarrow bool$
  **assumes**
    transp $(\prec_\mathsf{t})$ **and** asymp $(\prec_\mathsf{t})$ **and** totalp $(\prec_\mathsf{t})$ **and** wfp $(\prec_\mathsf{t})$ **and**
    $\forall \kappa :: 'f\ gctxt.\ \forall t_1\ t_2.\ t_1 \prec_\mathsf{t} t_2 \longrightarrow \kappa\langle t_1\rangle \prec_\mathsf{t} \kappa\langle t_2\rangle$ **and**
    $\forall \kappa :: 'f\ gctxt.\ \forall t.\ \kappa \neq \Box \longrightarrow t \prec_\mathsf{t} \kappa\langle t\rangle$

In Isabelle, a locale [8] consists of parameters (here, $\prec_\mathsf{t}$) that may depend on type variables (here, $'f$) paired with assumptions. Locales are a useful structuring mechanism. They allow us to declare parameters and assumptions once and reuse them in multiple related definitions and lemmas. When we later instantiate a locale, we must supply concrete arguments for the types and parameters and then discharge the proof obligations corresponding to the assumptions. This methodology emphasizes the modular nature of the proof development. Section 2.5 has a more complete introduction to locales.

The locale ground_ordering assumes that the binary relation $\prec_\mathsf{t}$ is a well-founded total ordering, is compatible with ground term contexts, and has the subterm property.

Inside the locale context, we lift the term ordering and its properties to literals ($\prec_\mathsf{lit}$) and clauses ($\prec_\mathsf{cls}$). We also configure the little-known order Isabelle proof method [108], a decision procedure for the quantifier-free theory of partial and total orderings, so that it can solve problems for our orderings.

Next, we define the notion of selection function:

**locale** select =
  **fixes** $sel :: 'a\ clause \Rightarrow 'a\ clause$
  **assumes**

$\forall C.\ sel\ C \subseteq C$ **and**
$\forall C.\ \forall L \in sel\ C.\ \mathsf{is\_neg}\ L$

The locale select fixes a function *sel* for clauses with any atom type $'a$. In this section, we instantiate $'a$ with $'f\ gatom$; Section 4.5 will instantiate it with its own atom type. Our assumptions on a selection function are that it always returns a submultiset of the argument $C$ and only returns negative literals.

We can now assemble the parameters and assumption for the ground calculus:

**locale** ground_superposition_calculus = ground_ordering $(\prec_\mathsf{t})$ + select $sel_\mathsf{G}$
**for**
$(\prec_\mathsf{t}) :: 'f\ gterm \Rightarrow 'f\ gterm \Rightarrow bool$ **and**
$sel_\mathsf{G} :: 'f\ gatom\ clause \Rightarrow 'f\ gatom\ clause$ +
**assumes** $\forall R :: ('f\ gterm \times 'f\ gterm)\ set.$ ground_critical_pair_theorem $R$

The locale ground_superposition_calculus extends both ground_ordering and select, inheriting all their assumptions as well as the definitions and theorems from their locale contexts. The parameters $\prec_\mathsf{t}$ and $sel_\mathsf{G}$ are provided with type annotations to control the instantiation of the type parameters ($'f$ in ground_ordering and $'a$ in select). We also assume that the critical pair theorem [2, Theorem 6.2.4] holds for ground terms. As a sanity check, we proved this theorem in Isabelle by adapting a similar, but license-incompatible, result from the `IsaFoR` project [110]. The `IsaFoR` formalization is published under the GNU Lesser General Public License (LGPL), which requires any derived work to be published under a compatible license. However, we whish to submit our formalization to the *AFP* under the BSD license found at `https://www.isa-afp.org/LICENSE`, which is not compatible. Assuming the theorem in the locale allows us to use the theorem in this formalization without providing the proof in the same *AFP* entry. We are in discussion with members of the `IsaFoR` project to find a place to publish our proof (the derived work) under the LGPL. Once the theorem is available in a license-compatible way, we will be able to alter our formalization to use the proved theorem directly and drop the assumption.

We can now specify the ground version of the inference rules presented in Section 4.2, using inductive predicates. Compared with their nonground counterparts, the ground rules benefit from two simplifications. First, neither renamings nor unifiers are needed because ground terms contain no variables. Second, terms and clauses can be compared directly using $\prec_\mathsf{t}$ and $\prec_\mathsf{cls}$ instead of using a reversed negated form since the orderings are total.

The ground superposition rule is as follows. The rule notation below defines an inductive predicate ground_superposition $D\ E\ C$ with the rule's premises $D, E$ as assumptions and the rule's conclusion $C$ as conclusion:

$$\frac{\overbrace{t \approx t' \vee D'}^{D} \qquad \overbrace{\kappa[t] \bowtie u \vee E'}^{E}}{\underbrace{\kappa[t'] \bowtie u \vee D' \vee E'}_{C}}\ \mathsf{ground\_superposition}\ D\ E\ C$$

Side conditions:

1. $\bowtie \in \{\approx, \not\approx\}$

2. $D \prec_{\mathsf{cls}} E$
3. $t' \prec_{\mathsf{t}} t$
4. $u \prec_{\mathsf{t}} \kappa[t]$
5. if $\bowtie = \approx$, then $sel_{\mathsf{G}}\ E = \{\}$ and $\kappa[t] \bowtie u$ is strictly maximal in $E$
6. if $\bowtie = \not\approx$, then $sel_{\mathsf{G}}\ E = \{\}$ and $\kappa[t] \bowtie u$ is maximal in $E$ or $\kappa[t] \bowtie u$ is maximal in $sel_{\mathsf{G}}\ E$
7. $sel_{\mathsf{G}}\ D = \{\}$
8. $t \approx t'$ is strictly maximal in $D$

The ground equality resolution rule is as follows:

$$\frac{\overbrace{t \not\approx t \vee D'}^{D}}{\underbrace{D'}_{C}}\ \mathsf{ground\_eq\_resolution}\ D\ C$$

Side conditions:

1. $sel_{\mathsf{G}}\ D = \{\}$ and $t \not\approx t$ is maximal in $D$ or $t \not\approx t$ is maximal in $sel_{\mathsf{G}}\ D$

The ground equality factoring rule is as follows:

$$\frac{\overbrace{t \approx t' \vee t \approx t'' \vee D'}^{D}}{\underbrace{t' \not\approx t'' \vee t \approx t'' \vee D'}_{C}}\ \mathsf{ground\_eq\_factoring}\ D\ C$$

Side conditions:

1. $sel_{\mathsf{G}}\ D = \{\}$
2. $t \approx t'$ is maximal in $D$
3. $t' \prec_{\mathsf{t}} t$

Following the structure required by the saturation framework, we define an inference system $\mathsf{Inf_G}$ and a consequence relation $\mathsf{entails_G}$. For formulas, we use the type of ground clauses *'f gclause*, and for contradictions, we use the empty clause $\bot$.

**Definition 4.2.** *The constant* $\mathsf{Inf_G} :: {}'f\ gclause\ inference\ set$ *represents all inferences of the ground superposition calculus:*

$$\begin{aligned}\mathsf{Inf_G} = {} & \{\langle [D, E]; C\rangle \mid \mathsf{ground\_superposition}\ D\ E\ C\} \cup \\ & \{\langle [D]; C\rangle \mid \mathsf{ground\_eq\_resolution}\ D\ C\} \cup \\ & \{\langle [D]; C\rangle \mid \mathsf{ground\_eq\_factoring}\ D\ C\}\end{aligned}$$

The consequence relation from Section 3.4 considers an interpretation to be a set of true atoms. Our ground atoms being unordered pairs of ground terms, our interpretations should be sets of *unordered* pairs. However, because Isabelle makes it easier to manipulate sets of *ordered* pairs, we use these as our interpretation and define a small wrapper with the help of the function $\mathsf{uprod} :: {}'a \times {}'a \Rightarrow {}'a\ uprod$ to bridge the gap.

**Definition 4.3.** *The predicates* $(\Vdash_{\mathsf{cls}}) :: ('f\ gterm \times 'f\ gterm)\ set \Rightarrow 'f\ gclause \Rightarrow bool$ *and* $(\Vdash_{\mathsf{clss}}) :: ('f\ gterm \times 'f\ gterm)\ set \Rightarrow 'f\ gclause\ set \Rightarrow bool$ *express that an interpretation models a clause and a clause set, respectively:*

$$\forall \mathcal{I}\ C.\ \mathcal{I} \Vdash_{\mathsf{cls}} C \longleftrightarrow \{\mathsf{uprod}\ r \mid r \in \mathcal{I}\} \models_{\mathsf{cls}} C$$

$$\forall \mathcal{I}\ N.\ \mathcal{I} \Vdash_{\mathsf{clss}} N \longleftrightarrow \{\mathsf{uprod}\ r \mid r \in \mathcal{I}\} \models_{\mathsf{clss}} N$$

We cannot use arbitrary sets of pairs as interpretations because the pairs should represent term equality. This means that a valid interpretation must behave like an equality relation. Specifically, we require a valid interpretation to be congruence relations on term contexts. This means that they must fulfill two requirements. First, a valid interpretation has to be an equivalence relation, i.e., reflexive, symmetric, and transitive. Second, a valid interpretation has to be compatible with ground context application. We encode these requirements in the $\mathsf{entails}_{\mathsf{G}}$ predicate below.

**Definition 4.4.** *The predicate* compatible_with_gctxt $:: ('f\ gterm \times 'f\ gterm)\ set \Rightarrow$ *bool expresses that all term pairs considered equals are compatible with ground context application:*

$$\begin{aligned}&\forall \mathcal{I}.\ \mathsf{compatible\_with\_gctxt}\ \mathcal{I} \longleftrightarrow\\ &\quad (\forall \kappa :: 'f\ gctxt.\ \forall t\ t'.\ \langle t; t'\rangle \in \mathcal{I} \longrightarrow \langle \kappa[t]; \kappa[t']\rangle \in \mathcal{I})\end{aligned}$$

**Definition 4.5.** *The predicate* $\mathsf{entails}_{\mathsf{G}} :: 'f\ gclause\ set \Rightarrow 'f\ gclause\ set \Rightarrow bool$ *expresses that a clause set entails another clause set, i.e., every valid interpretation of former is also a valid interpretation of the later:*

$$\begin{aligned}&\forall N_1\ N_2.\ \mathsf{entails}_{\mathsf{G}}\ N_1\ N_2 \longleftrightarrow\\ &\quad (\forall \mathcal{I} :: ('f\ gterm \times 'f\ gterm)\ set.\\ &\qquad \mathsf{refl}\ \mathcal{I} \longrightarrow \mathsf{sym}\ \mathcal{I} \longrightarrow \mathsf{trans}\ \mathcal{I} \longrightarrow \mathsf{compatible\_with\_gctxt}\ \mathcal{I} \longrightarrow\\ &\qquad \mathcal{I} \Vdash_{\mathsf{clss}} N_1 \longrightarrow \mathcal{I} \Vdash_{\mathsf{clss}} N_2)\end{aligned}$$

Equipped with $\mathsf{Inf}_{\mathsf{G}}$ and $\mathsf{entails}_{\mathsf{G}}$, we can start to use the saturation framework. As a sanity check, we first instantiate the sound_inference_system locale to make sure that the ground superposition calculus is sound and that our definitions correspond to what the framework expects:

**sublocale** ground_superposition_calculus $\subseteq$ sound_inference_system **where**
  $\mathsf{Inf} = \mathsf{Inf}_{\mathsf{G}}$ **and** $\mathsf{Bot} = \{\bot\}$ **and** $\mathsf{entails} = \mathsf{entails}_{\mathsf{G}}$

The sublocale notation means that definitions and theorems from ground_superposition_calculus are sufficient to prove the assumptions of sound_inference_system w.r.t. the given parameter instantiations. At this point, Isabelle requires us to actually prove the assumptions. The soundness proof of the inference system amounts to proving the soundness of each rule, i.e., that the conclusion of each rule is entailed by its premises.

As the redundancy criterion, we reuse the standard redundancy criterion defined in the Isabelle saturation framework [21]:

**sublocale** ground_superposition_calculus ⊆
  calculus_with_finitary_standard_redundancy **where**
Inf = $\mathsf{Inf}_\mathsf{G}$ **and** Bot = $\{\bot\}$ **and** entails = $\mathsf{entails}_\mathsf{G}$ **and** less = $(\prec_\mathsf{cls})$
**defines** $\mathsf{Red}_\mathsf{IG}$ = $\mathsf{Red}_\mathsf{I}$ **and** $\mathsf{Red}_\mathsf{FG}$ = $\mathsf{Red}_\mathsf{F}$

The locale calculus_with_finitary_standard_redundancy defines two functions that we want to reuse: $\mathsf{Red}_\mathsf{I} :: 'f\,gclause\,set \Rightarrow 'f\,gclause\,inference\,set$, identifying redundant inferences, and $\mathsf{Red}_\mathsf{F} :: 'f\,gclause\,set \Rightarrow 'f\,gclause\,set$, identifying redundant formulas. We rename them to $\mathsf{Red}_\mathsf{IG}$ and $\mathsf{Red}_\mathsf{FG}$, respectively.

To prove refutational completeness, we will exhibit a valid interpretation for a given saturated clause set. We build this interpretation by defining a confluent and terminating set of rewrite rules $R_\infty$, which we lift to an interpretation $[\![R_\infty]\!]^\downarrow$ that defines term equality. Each rewrite rule is a pair $\langle t; t'\rangle$, written $t \to t'$.

**Definition 4.6.** *The function* $[\![\cdot]\!] :: ('f\,gterm \times 'f\,gterm)\,set \Rightarrow ('f\,gterm \times 'f\,gterm)\,set$ *expands a rewrite rule set to all term contexts:*

$$\forall R.\ [\![R]\!] = \{\kappa[t] \to \kappa[t'] \mid t \to t' \in R\}$$

**Definition 4.7.** *The function* $\cdot^\downarrow :: ('f\,gterm \times 'f\,gterm)\,set \Rightarrow ('f\,gterm \times 'f\,gterm)\,set$ *produces the set of all term pairs considered equal w.r.t. a set of rewrite rules:*

$$\forall R.\ R^\downarrow = \{t \to t' \mid \exists t''.\ t \to t'' \in R^* \wedge t' \to t'' \in R^*\}$$

Two terms are considered equal if they are joinable, i.e., if they have a common reduct.

Now that we can lift a set of rewrite rules to a model, we define two mutually recursive functions that construct such a set for a given clause set.

**Definition 4.8.** *Let* $N^{\prec_\mathsf{cls} D} = \{C \in N \mid C \prec_\mathsf{cls} D\}$ *for any clause set* $N$ *and clause* $D$. *The mutually recursive functions* $\mathsf{epsilon} :: 'f\,gclause\,set \Rightarrow 'f\,gclause \Rightarrow ('f\,gterm \times 'f\,gterm)\,set$ *and* $\mathsf{rewrite\_sys} :: 'f\,gclause\,set \Rightarrow ('f\,gterm \times 'f\,gterm)\,set$ *generate a term rewriting system for a given clause set:*

$$\begin{aligned}\forall N\ C.\ \mathsf{epsilon}\ N\ C = \{t \to t' \mid \exists C'.\ & C \in N \wedge sel_\mathsf{G}\ C = \{\} \wedge\\ & C = (t \approx t' \vee C') \wedge\\ & t \approx t' \textit{ is strictly maximal in } C \wedge t' \prec_\mathsf{t} t \wedge\\ & [\![\mathsf{rewrite\_sys}\ N^{\prec_\mathsf{cls} C}]\!]^\downarrow \not\models_\mathsf{cls} C \wedge\\ & [\![\mathsf{rewrite\_sys}\ N^{\prec_\mathsf{cls} C} \cup \{t \to t'\}]\!]^\downarrow \not\models_\mathsf{cls} C' \wedge\\ & t \textit{ is in normal form w.r.t. } [\![\mathsf{rewrite\_sys}\ N^{\prec_\mathsf{cls} C}]\!]^\downarrow\}\end{aligned}$$

$$\forall N.\ \mathsf{rewrite\_sys}\ N = \bigcup_{C \in N} \mathsf{epsilon}\ N\ C$$

In Isabelle, we first defined epsilon as a recursive function where all occurrences of rewrite_sys are replaced by their definition, i.e., recursive but not mutually recursive. We then defined rewrite_sys as a nonrecursive function. Finally, we proved the first equality shown in Definition 4.8 and used it in the rest of the formalization.

We reuse the definitions of joinability ($\cdot^\downarrow$) and of normal form (i.e., irreducibility) from an existing formalization of abstract rewriting systems [106].

The model construction iterates over the clause set, starting from the smallest clause following the ordering $\prec_{\mathsf{cls}}$, and collects a set of rewrite rules. At any point, we can use $[\![\cdot]\!]^{\downarrow}$ to obtain the candidate model. At each iteration, epsilon returns a set of rewrite rules that are added to the term rewriting system: Either the considered clause is already true w.r.t. the candidate model, in which case epsilon returns the empty set, or epsilon returns a single new rewrite rule that should make the clause true. We call a clause *productive* if epsilon produces a new rewrite rule. Note that the produced rule is unique (i.e., $\forall N\ C.\ |\mathsf{epsilon}\ N\ C| \leq 1$) because the strictly maximal literal in a clause w.r.t. the total ordering $\prec_{\mathsf{lit}}$ (on line two of epsilon's definition) is unique. This is the reason why we refer to "the" produced atom.

**Example 4.9.** *Let* $\mathsf{f}$ *be a function symbol. Let* $\mathsf{a}$, $\mathsf{b}$, $\mathsf{c}$, $\mathsf{d}$, *and* $\mathsf{e}$ *be ground terms. Assume* $\prec_{\mathsf{t}}$ *is the lexicographic path ordering with the precedence* $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{d} \prec \mathsf{e} \prec \mathsf{f}$. *Consider the clause set* $N = \{\underline{\mathsf{d} \approx \mathsf{c}},\ \mathsf{b} \approx \mathsf{a} \vee \underline{\mathsf{e} \not\approx \mathsf{c}},\ \mathsf{b} \not\approx \mathsf{b} \vee \underline{\mathsf{f}(\mathsf{b}) \approx \mathsf{a}},\ \underline{\mathsf{f}(\mathsf{c}) \approx \mathsf{b}},\ \mathsf{f}(\mathsf{b}) \approx \mathsf{a} \vee \underline{\mathsf{f}(\mathsf{c}) \not\approx \mathsf{b}},\ \mathsf{f}(\mathsf{b}) \approx \mathsf{a} \vee \underline{\mathsf{f}(\mathsf{d}) \not\approx \mathsf{b}}\}$ *saturated w.r.t. the ground superposition calculus. The underlined literals are maximal in their respective clause. The following table shows the result of each iteration of the model construction:*

| *Iteration* | *Clause* $C$ | rewrite_sys $N^{\prec_{\mathsf{cls}} C}$ | epsilon $N\ C$ |
|---|---|---|---|
| *1* | $\underline{\mathsf{d} \approx \mathsf{c}}$ | $\{\}$ | $\{\mathsf{d} \to \mathsf{c}\}$ |
| *2* | $\mathsf{b} \approx \mathsf{a} \vee \underline{\mathsf{e} \not\approx \mathsf{c}}$ | $\{\mathsf{d} \to \mathsf{c}\}$ | $\{\}$ |
| *3* | $\mathsf{b} \not\approx \mathsf{b} \vee \underline{\mathsf{f}(\mathsf{b}) \approx \mathsf{a}}$ | $\{\mathsf{d} \to \mathsf{c}\}$ | $\{\mathsf{f}(\mathsf{b}) \to \mathsf{a}\}$ |
| *4* | $\underline{\mathsf{f}(\mathsf{c}) \approx \mathsf{b}}$ | $\{\mathsf{d} \to \mathsf{c},\ \mathsf{f}(\mathsf{b}) \to \mathsf{a}\}$ | $\{\mathsf{f}(\mathsf{c}) \to \mathsf{b}\}$ |
| *5* | $\mathsf{f}(\mathsf{b}) \approx \mathsf{a} \vee \underline{\mathsf{f}(\mathsf{c}) \not\approx \mathsf{b}}$ | $\{\mathsf{d} \to \mathsf{c},\ \mathsf{f}(\mathsf{b}) \to \mathsf{a},\ \mathsf{f}(\mathsf{c}) \to \mathsf{b}\}$ | $\{\}$ |
| *6* | $\mathsf{f}(\mathsf{b}) \approx \mathsf{a} \vee \underline{\mathsf{f}(\mathsf{d}) \not\approx \mathsf{b}}$ | $\{\mathsf{d} \to \mathsf{c},\ \mathsf{f}(\mathsf{b}) \to \mathsf{a},\ \mathsf{f}(\mathsf{c}) \to \mathsf{b}\}$ | $\{\}$ |

*At each iteration* $i + 1$*, the term rewriting system consists of the union of the term rewriting system of iteration* $i$ *and the "epsilon" of iteration* $i$*. As expected, the interpretation after iteration 6 is a model of* $N$*.*

The conditions on rewrite rule production were chosen so that the resulting term rewriting system is confluent. Specifically, we prove strong normalization and the weak Church–Rosser property, which together imply the Church–Rosser property, which is equivalent to confluence. We refer the interested reader to *Term Rewriting and All That*, by Baader and Nipkow [2], for more details on the Church–Rosser property.

**Lemma 4.10.** *Let* $N$ *be a ground clause set. The term rewriting system* $[\![\mathsf{rewrite\_sys} N]\!]$ *is strongly normalizing.*

**Lemma 4.11.** *Let* $N$ *be a ground clause set. The term rewriting system* $[\![\mathsf{rewrite\_sys} N]\!]$ *has the weak Church–Rosser property.*

This enables us to prove that the resulting model is a valid interpretation.

**Lemma 4.12.** *Let* $N$ *be a ground clause set. The term rewriting system* $[\![\mathsf{rewrite\_sys} N]\!]^{\downarrow}$ *can be interpreted as a equivalence relation (i.e., it is reflexive, symmetric, transitive) that is compatible with ground contexts.*

Now that we can build a valid interpretation for a clause set, it remains to show that it satisfies all clauses from this set. We first need a pair of lemmas that express monotonicity properties of the construction:

**Lemma 4.13.** *Let $N$ be a ground clause set and $C$ be a ground clause. Let $t$ and $t'$ be ground terms. If* epsilon $N\ C = \{t \to t'\}$*, then*

1. $[\![\mathsf{rewrite\_sys}\ N]\!]^{\downarrow} \models_{\mathsf{cls}} C$;
2. $\forall D \in N.\ C \prec_{\mathsf{cls}} D \longrightarrow [\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} D}]\!]^{\downarrow} \models_{\mathsf{cls}} C$;
3. $[\![\mathsf{rewrite\_sys}\ N]\!]^{\downarrow} \not\models_{\mathsf{cls}} C - \{t \approx t'\}$; *and*
4. $\forall D \in N.\ C \prec_{\mathsf{cls}} D \longrightarrow [\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} D}]\!]^{\downarrow} \not\models_{\mathsf{cls}} C - \{t \approx t'\}$.

**Lemma 4.14.** *Let $N$ be a ground clause set and $C \in N$ be a ground clause. If $[\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} C}]\!]^{\downarrow} \models_{\mathsf{cls}} C$, then*

1. $[\![\mathsf{rewrite\_sys}\ N]\!]^{\downarrow} \models_{\mathsf{cls}} C$ *and*
2. $\forall D \in N.\ C \prec_{\mathsf{cls}} D \longrightarrow [\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} D}]\!]^{\downarrow} \models_{\mathsf{cls}} C$.

We can now prove that our model construction works for all clauses.

**Lemma 4.15.** *Let $N$ be a saturated ground clause set and $C \in N$ be a ground clause. If $\bot \notin N$, then*

1. epsilon $N\ C = \{\} \longleftrightarrow [\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} C}]\!]^{\downarrow} \models_{\mathsf{cls}} C$ *and*
2. $\forall D \in N.\ C \prec_{\mathsf{cls}} D \longrightarrow [\![\mathsf{rewrite\_sys}\ N^{\prec_{\mathsf{cls}} D}]\!]^{\downarrow} \models_{\mathsf{cls}} C$.

*Proof Sketch.* By well-founded induction w.r.t. $\prec_{\mathsf{cls}}$. □

**Lemma 4.16** (Ground Model Construction)**.** *Let $N$ be a saturated ground clause set and $C \in N$ be a ground clause. If $\bot \notin N$, then $[\![\mathsf{rewrite\_sys}\ N]\!]^{\downarrow} \models_{\mathsf{cls}} C$.*

*Proof Sketch.* This follows from Lemmas 4.14 and 4.15 if epsilon $N\ C = \{\}$ and from Lemma 4.13 otherwise. □

**Theorem 4.17** (Ground Refutational Completeness)**.** *Let $N$ be a saturated ground clause set. If* $\mathsf{entails_G}\ N\ \{\bot\}$*, then $\bot \in N$.*

*Proof Sketch.* By contraposition, we assume $\bot \notin N$ and show $\neg\ \mathsf{entails_G}\ N\ \{\bot\}$. We must show the existence of an interpretation $\mathcal{I}$ such that **(1)** $\mathcal{I}$ is reflexive, symmetric, transitive, and compatible with ground contexts; **(2)** $\mathcal{I} \models N$; and **(3)** $\mathcal{I} \not\models_{\mathsf{cls}} \bot$. We choose $\mathcal{I} = [\![\mathsf{rewrite\_sys}\ N]\!]^{\downarrow}$. Step 1 follows from Lemma 4.12. Step 2 follows from Lemma 4.16. Step 3 follows from the definition of $\models_{\mathsf{cls}}$. □

Finally, we can provide our main result for the ground calculus by instantiating the locale statically_complete_calculus from the saturation framework:

**sublocale** ground_superposition_calculus $\subseteq$ statically_complete_calculus **where**
$\mathsf{Inf} = \mathsf{Inf_G}$ **and** $\mathsf{Bot} = \{\bot\}$ **and** $\mathsf{entails} = \mathsf{entails_G}$ **and** $\mathsf{less} = (\prec_{\mathsf{cls}})$ **and**
$\mathsf{Red_I} = \mathsf{Red_{IG}}$ **and** $\mathsf{Red_F} = \mathsf{Red_{FG}}$

We use Theorem 4.17 to discharge the proof obligation.

## 4.5 The Nonground Proof

On the nonground level, we define a locale first_order_superposition_calculus for the nonground calculus parametrized by an ordering on nonground terms, a selection function for nonground clauses, and a family of tiebreakers used to implement subsumption. In this locale, we define the three inference rules for superposition, equality resolution, and equality factoring. These rules are significantly more complex than their ground counterparts, as they require renamings and unifiers to handle variables. They also need to cope with the fact that the nonground term ordering is partial.

We specify under which conditions a ground selection function (a selection function on ground clauses) is compatible with our selection function on nonground clauses. We then use the lifting_intersection locale of the saturation framework to lift a family of ground calculi indexed by compatible ground selection functions to the nonground level. This provides us with a lifted entailment relation and a lifted redundancy criterion.

We then prove that nonground inferences from a clause set overapproximate the ground inferences from the ground instances of said clause set.

Using this, we instantiate the statically_complete_calculus and discharge the proof obligations. We have verified the static refutational completeness of first-order superposition and the saturation frameworks provides us with a proof of dynamic refutational completeness.

To ensure that there are no inconsistencies in our locale assumptions, we instantiate the locales with the Knuth–Bendix ordering, a trivial selection function, and a trivial tiebreakers family.

## 4.6 Related Work

The saturation framework [119] has been used in the completeness proof of several new variants of superposition:

- Boolean $\lambda$-superposition [14] for higher-order logic, as well as its predecessors Boolean-free $\lambda$-superposition [15] and Boolean-free $\lambda$-free superposition [13] that operate on fragments of higher-order logic.
- superposition with delayed unification [17] for first-order logic, which adds constraints to the conclusions of inferences instead of performing full unifications.

An extended abstract by Tourret [112] discusses how these use the framework. The work described in the present chapter could serve as a foundation to formalize these proofs.

The Isabelle/HOL formalization of the saturation framework was introduced together with an instance of the resolution calculus and an abstract resolution prover called RP by Tourret and Blanchette [113]. Other theorem proving techniques formalized in Isabelle/HOL include an executable SAT solver by Blanchette et al. [19] based on CDCL (conflict-driven clause learning) for propositional logic with state-of-the-art optimizations, a sequent calculus by From, Blackburn, and Villadsen [54], a tableau calculus by From, Schlichtkrull, and Villadsen [55], and another version of resolution and RP by Schlichtkrull et al. [97] following Bachmair and Ganzinger's original, more ad hoc proof that was extended to an executable prover [98]. Most

recently, the newly created SCL calculus [52], which follows a CDCL-like approach to theorem proving in first-order logic, was also verified in Isabelle/HOL by Bromberger, Desharnais, and Weidenbach [25] as it was being developed; this formalization work is presented in Chapter 5. Also relevant here is Paulson's formalization of Gödel's incompleteness theorems [87, 88].

Isabelle/HOL is possibly the most widely used system for formalizing automated reasoning results, but other proof assistants are used as well. Early results include Shankar's proof of Gödel's first incompleteness theorem in Nqthm [103], Persson's completeness proof for intuitionistic predicate logic in ALF [91], and Harrison's formalization of basic first-order model theory in HOL Light [62]. We refer to Blanchette [18, Section 5] for a survey.

Finally, the work closest to ours, already mentioned in the introduction, is the formalization of a variant of the superposition calculus in Isabelle/HOL by Peltier [90]. Our initial intent was to integrate his calculus with the saturation framework, but after months of fruitless attempts, we decided to start from scratch, which resulted in the present work.

A first obstacle we encountered was related to Peltier's redundancy criterion. He relies on a notion that is sufficient to prove static refutational completeness but cannot be lifted to dynamic completeness because his redundancy is defined in terms of smaller or equal clauses rather than strictly smaller clauses. This makes it unsuitable for use in the saturation framework, but we managed to replace it with a suitable criterion without changing the calculus, allowing us to pursue our work in this direction for a while.

What made us switch approach was an incompatibility requiring a major modification of Peltier's formalization itself. Peltier works with closures, i.e., pairs $\langle C; \sigma \rangle$ consisting of a *set* of literals $C$ and a substitution $\sigma$. This calculus is defined directly on the nonground level, where static completeness is proved. For integration into the saturation framework, we wanted a ground version of the calculus, which we obtained by restricting the substitutions to groundings only and operating on clauses as sets of ground literals $C\sigma$. However, this made it impossible to overapproximate this calculus with Peltier's calculus on the nonground level, which is needed for the lifting to be possible in the framework. The issue is that we do not want to match a literal $K$ in a ground clause to two literals $L_1, L_2$ in a nonground closure $\langle L_1 \vee L_2 \vee C; \sigma \rangle$ such that $L_1\sigma = L_2\sigma = K$, because this breaks the lifting. Fixing this would require working directly on closures also at the ground level, but for completeness, such a calculus would need to allow superposition inferences also in the range of the substitution. A new proof of ground refutational completeness would have had to be provided for this new calculus. It seemed more convenient to formalize a calculus operating on *multisets* of literals instead of closures, especially given that the Isabelle multiset library was already well developed for use in theorem proving formalization.

Our formalization consists of approximately 12 000 nonblank lines[2], 7000 of which are for nonbackground theories. For comparison, Peltier's formalization consists of approximately 9000 nonblank lines, 7000 of which are for nonbackground theories. All numbers are rounded to the nearest thousand. Interestingly, the two formalizations have approximately the same size even though they are written in very different styles. To our surprise, the additional modularity of our work did not lead to a

[2]Counted using `grep -Ev '^[[:blank:]]*$'`.

shorter proof.

## 4.7 Conclusion

We restructured the refutational completeness proof of superposition using the saturation framework. We first proved refutational completeness for the ground calculus and lifted the proof to the full, nonground calculus. Next, we formalized this pen-and-paper proof in Isabelle/HOL. The formalization can be seen as a case study for the `IsaFoR` library and the saturation framework, as well as for basic Isabelle tools such as locales, which facilitate modularity and proof reuse.

We see three main directions for future work. First, the proof could be extended to support simply typed or rank-1-polymorphic first-order terms. Second, the completeness proofs of variants of superposition, such as hierarchic superposition [6, 11], combinatory superposition [16], and $\lambda$-superposition [14], could be formalized as well. Third, the formalization of superposition (or that of variants) could be extended to obtain a verified executable prover.

# Chapter 5

# SCL(FOL): Simple Clause Learning for First-Order Logic

This chapter is based on a conference paper coauthored with Martin Bromberger and Christoph Weidenbach [25]. Section 5.3 describes my formalization work.

## 5.1 Introduction

The SCL ("Clause Learning from Simple Models" or simply "Simple Clause Learning") family of calculi lifts a conflict-driven clause learning (CDCL) approach [12, 82] to first-order logic: SCL(FOL) is for first-order logic without equality [29, 52], SCL(T) is for first-order logic with theories [26], SCL(EQ) is for first-order logic with equality [72], and HSCL is for exhaustive partial models exploration in first-order logic without equality [28]. In its original formulation by Fiori and Weidenbach [52], SCL(FOL) required exhaustive propagation and a precise strategy for the application of the rules in order to learn nonredundant clauses. This was improved upon by Bromberger, Fiori, and Weidenbach [26] in SCL(T), which dropped exhaustive propagation and weakened the strategy (i.e., any run according to the strategy in [52] is also a run according to the strategy in [26]). The version of SCL(FOL) presented later by Bromberger, Schwarz, and Weidenbach [29] integrates those changes and additionally refines the Backtrack rule.

We present an Isabelle/HOL formalization of the (nonexecutable) specification of the SCL(FOL) calculus based on an early version of Bromberger et al.'s paper and developed in parallel to the final version. The main results are soundness, nonredundancy of learned clauses, termination, and refutational completeness. In contrast to the goal of Bromberger et al. to guide toward an implementation, our goal is to be as simple and general as possible. For that, we (i) simplified the calculus (e.g., no more explicitly tracking of decision levels), (ii) generalized the calculus (e.g., multiple acceptable positions in the Backtrack rule), (iii) strengthened existing theorems (e.g., Theorem 5.11 on nonredundancy), and (iv) proved new theorems (e.g., Corollary 5.12 on nonsubsumption).

This work is part of the IsaFoL (Isabelle Formalization of Logic) effort [18], which aims at developing a library of results about logical calculi. The Isabelle theory files are available in the *Archive of Formal Proofs* (AFP) [40] and amount to approximately

10 000 nonblank lines of source text[1]. They build heavily upon several other entries of the AFP: (i) `First_Order_Terms` [107] for first-order terms, term substitutions, and MGU; (ii) `Ordered_Resolution_Prover` [97, 99, 100] for the clausal calculus, clause substitutions, Herbrand interpretation, and compactness of first-order logic; and (iii) `Saturation_Framework_Extensions` [21, 119] for entailment of the clausal calculus. We contributed many lemmas and definitions back to both the Isabelle distribution and the aforementioned AFP entries (e.g., over 50 to `First_Order_Terms`). We made heavy use of the Isar language [123] to write structured proofs, the Sledgehammer tool [89] for proof automation, and locales [8]—Isabelle's parameterized module system—to structure our development and reuse existing components from the AFP entries.

The formalization follows the basic ideas of existing formalizations of the first-order resolution calculus [97] and propositional CDCL calculi [19, 20]. Compared with propositional logic, first-order logic adds a number of challenges: the extra term level requires to consider variables, substitutions, groundings, and the concept of factorization. To preserve completeness, propagation of ground literals must not be exhaustive anymore, resulting in a level-wise exploration w.r.t. a bounding atom. Inside this bound, the calculus always terminates. If one level does not suffice to find a refutation, the bound can be increased and exploration can be continued. For unsatisfiable formulas, we prove the existence of a bound sufficient to derive $\bot$, which guarantees that only finitely many levels need to be explored.

The chapter is organized as follows. Section 5.2 recaps the SCL(FOL) calculus from Bromberger et al. as the basis of our formalization presented in Section 5.3. We first present the Isabelle formalization of the abstract rules of the SCL(FOL) calculus. Then we prove invariants preserved by the rules starting from the initial state, Lemma 5.1. Subsequently, we prove soundness, Theorem 5.7, nonredundancy of learned clauses, Theorem 5.11, termination with respect to a fixed bound, Theorem 5.20, and finally refutational completeness with respect to an appropriate bound, Theorem 5.22. We discuss important aspects of the formalization and proof ideas here and refer the reader to the formalization for more details. We end with a short conclusion of the obtained results.

## 5.2 The SCL(FOL) Calculus

We briefly repeat basic notions regarding the SCL(FOL) calculus presented by Bromberger et al. We consider an untyped, first-order logic without equality. *Terms*, *clauses*, and *entailment* are defined in Chapter 3.

The strict ordering $\prec_B$ is total on ground literals and is such that each literal has finitely many lesser literals (i.e., $\forall \beta.\ \mathsf{finite}\ \{L \mid L \prec_B \beta\}$). An example of such an ordering could be the Knuth–Bendix ordering [69] without zero-weight symbols. Note that the lexicographic path ordering [2, Section 5.4.2] does not satisfy the last condition of a $\prec_B$ ordering although it is well-founded and total on ground terms.

An annotated literal is the pairing of a literal with an annotation. We call it a *decision literal* when the annotation is a natural number $n$ indicating the literal's level (i.e., that it is the $n$th decision) and a *propagation literal* when the annotation

[1] Counted using `grep -Ev '^[[:blank:]]*$'`.

is a closure of the clause the literal originated from. The literal of an annotated literal $\mathcal{K}$ is denoted $\mathsf{lit}\,\mathcal{K}$ and the annotation is denoted $\mathsf{ann}\,\mathcal{K}$. The level of a clause is the maximum level of its literals. A *trail* is a finite sequence of annotated ground literals; it grows from left to right. The empty trail is written $\epsilon$ and appending a new annotated literal $\mathcal{K}$ to a trail $\Gamma$ is written $\Gamma, \mathcal{K}$. The concatenation of two trails $\Gamma_1$ and $\Gamma_2$ is written $\Gamma_2, \Gamma_1$. A trail $\Gamma$ can be converted to a set of annotated literals with $\mathsf{set}\,\Gamma$.

A literal $L$ is true under trail $\Gamma$ if $L \in \{\mathsf{lit}\,\mathcal{K} \mid \mathcal{K} \in \mathsf{set}\,\Gamma\}$. A literal $L$ is false under trail $\Gamma$ if $\mathsf{comp}\,L \in \{\mathsf{lit}\,\mathcal{K} \mid \mathcal{K} \in \mathsf{set}\,\Gamma\}$. A literal $L$ is defined in a trail $\Gamma$ if $L$ is true or false under $\Gamma$; otherwise, it is undefined. A clause $C$ is true under trail $\Gamma$ if at least one of its literals is true (i.e., if $\exists L \in C.\ L$ is true under $\Gamma$). A clause $C$ is false under trail $\Gamma$ if all its literal are false (i.e., if $\forall L \in C.\ L$ is false under $\Gamma$). A clause $C$ is defined in a trail $\Gamma$ if all its literals are defined (i.e., if $\forall L \in C.\ L$ is defined in $\Gamma$); otherwise, it is undefined.

The SCL(FOL) calculus is defined as a transition system operating on states $\langle \Gamma; N; U; \beta; k; \mathcal{C}\rangle$ where $\Gamma$ is a trail, $N$ is a finite set of initial clauses, $U$ is a finite set of learned clauses, $\beta$ is a bounding atom restricting the considered ground literals, $k$ is a natural number counting the number of decision literals in $\Gamma$, and $\mathcal{C}$ is either $\dagger$ or a clause closure $\langle C; \gamma\rangle$ such that $C\gamma$ is ground and false in $\Gamma$. The initial state is $\langle \epsilon; N; \{\}; \beta; 0; \dagger\rangle$ for some initial clause set $N$ and bound $\beta$.

The transition relation $\Rightarrow_{\mathrm{SCL}}$ is a mapping between states. The rules below are from Bromberger et al.[2] and serve as a reference for the Isabelle formalization described in Section 5.3.

**Propagate** $\langle \Gamma; N; U; \beta; k; \dagger\rangle \Rightarrow_{\mathrm{SCL}} \langle \Gamma, L\gamma^{\langle (C_0 \vee L)\mu; \gamma\rangle}; N; U; \beta; k; \dagger\rangle$
Side conditions:

1. $(C \vee L) \in N \cup U$
2. $(C \vee L)\gamma$ is ground
3. $(C \vee L)\gamma \prec_B \{\beta\}$
4. $C = C_0 \vee C_1$
5. $C_1\gamma = L\gamma \vee \cdots \vee L\gamma$
6. $C_0\gamma$ does not contain $L\gamma$
7. $C_0\gamma$ is false under $\Gamma$
8. $L\gamma$ is undefined in $\Gamma$
9. $\mu$ is the IMGU of the literals in $C_1$ and $L$

**Decide** $\langle \Gamma; N; U; \beta; k; \dagger\rangle \Rightarrow_{\mathrm{SCL}} \langle \Gamma, L\gamma^{k+1}; N; U; \beta; k+1; \dagger\rangle$
Side conditions:

1. $(C \vee L) \in N \cup U$
2. $L\gamma$ is a ground literal
3. $L\gamma$ is undefined in $\Gamma$
4. $L\gamma \prec_B \beta$

**Conflict** $\langle \Gamma; N; U; \beta; k; \dagger\rangle \Rightarrow_{\mathrm{SCL}} \langle \Gamma; N; U; \beta; k; \langle C; \gamma\rangle\rangle$

[2]The syntax and naming were slightly changed for uniformity with the rest of this thesis.

Side conditions:

1. $C \in N \cup U$
2. $C\gamma$ is ground
3. $C\gamma$ is false under $\Gamma$

These rules construct a (partial) model via the trail $\Gamma$ for $N \cup U$ until a conflict (i.e., a clause false under $\Gamma$) is found. The above rules always terminate, because there are only finitely many ground literals $L$ with $L \prec_B \beta$. It might be necessary to successively increase $\beta$ for full refutational completeness.

**Skip** $\langle \Gamma, \mathcal{K}; N; U; \beta; k; \langle C; \gamma \rangle \rangle \Rightarrow_{\text{SCL}} \langle \Gamma; N; U; \beta; k - i; \langle C; \gamma \rangle \rangle$
Side conditions:

1. $\mathsf{comp}\,(\mathsf{lit}\,\mathcal{K})$ does not occur in $C\gamma$
2. if $\mathcal{K}$ is a decision literal, then $i = 1$; otherwise, $i = 0$

**Factorize** $\langle \Gamma; N; U; \beta; k; \langle C \vee L \vee L'; \gamma \rangle \rangle \Rightarrow_{\text{SCL}} \langle \Gamma; N; U; \beta; k; \langle (C \vee L)\mu; \gamma \rangle \rangle$
Side conditions:

1. $L\gamma = L'\gamma$
2. $\mu$ is the IMGU of the literals $L$ and $L'$

Note that this rule may be used multiple times if the conflicting clause contains more than two duplicates of a given literal or if multiple distinct literals have duplicates.

**Resolve** $\langle \Gamma, K\gamma_D^{\langle D \vee K; \gamma_D \rangle}; N; U; \beta; k; \langle C \vee L; \gamma_C \rangle \rangle \Rightarrow_{\text{SCL}}$
$\langle \Gamma, K\gamma_D^{\langle D \vee K; \gamma_D \rangle}; N; U; \beta; k; \langle (C \vee D)\mu; \gamma_C \circledast \gamma_D \rangle \rangle$
Side conditions:

1. $K\gamma_D = \mathsf{comp}\,(L\gamma_C)$
2. $\mu$ is the IMGU of the literals $K$ and $\mathsf{comp}\,L$

The clauses $D \vee K$ and $C \vee L$ are assumed to have disjoint variables.

**Backtrack** $\langle \Gamma_0, \mathcal{K}, \Gamma_1, (\mathsf{comp}\,(L\gamma))^k; N; U; \beta; k; \langle C \vee L; \gamma \rangle \rangle \Rightarrow_{\text{SCL}}$
$\langle \Gamma_0; N; U \cup \{C \vee L\}; \beta; j; \dagger \rangle$
Side conditions:

1. $C\gamma$ is of level $i' < k$
2. $\Gamma_0, \mathcal{K}$ is the minimal trail subsequence such that there is a grounding substitution $\gamma'$ with $(C \vee L)\gamma'$ false under $\Gamma_0, \mathcal{K}$ but not under $\Gamma_0$
3. $\Gamma_0$ is of level $j$

The clause $C \vee L$ added by the rule Backtrack to $U$ is called a *learned clause*. The empty clause $\bot$ can only be generated by the rule Resolve or be already present in $N$. Hence, as usual for CDCL-style calculi, the generation of $\bot$, together with the clauses in $N \cup U$, represents a resolution refutation.

A sequence of SCL rule applications is called a *reasonable run* if the rule Decide does not enable an immediate application of rule Conflict. A sequence of SCL rule applications is called a *regular run* if it is a reasonable run and the rule Conflict has precedence over all other rules.

## 5.3 Formalization of the SCL(FOL) Calculus

The formalization introduces some new concepts absent from Section 5.2. A multiset $C$ can be converted to a set, i.e., without duplicates, with $\mathsf{set}\,C$. The cardinality of a multiset—the sum of the multiplicities of its elements—is denoted by $|C|$. The multiset whose only element is $x$ with multiplicity $n$ is denoted by $\mathsf{repeat}\,n\,x$; note that $\forall n\,x.\ \mathsf{count}\,(\mathsf{repeat}\,n\,x)\,x = n$ and $\forall n\,x.\ n > 0 \longrightarrow \mathsf{set}\,(\mathsf{repeat}\,n\,x) = \{x\}$. The *adaptation* of a substitution $\sigma$ to a renaming $\rho$ is a function whose domain is the renamed domain of $\sigma$ and whose codomain is the same as $\sigma$; it is defined as the function $(\lambda x.\ \mathsf{if}\,x \in \{y\rho \mid y \in \mathsf{dom}\,\sigma\}\,\mathsf{then}\,(x\rho^{-1})\sigma\,\mathsf{else}\,x)$. A substitution $\gamma$ is a *merged grounding* of a grounding $\gamma_A$ for a variable set $A$ and a grounding $\gamma_B$ for variable set $B$ if $(A \cap B = \{\} \longrightarrow (\forall x \in A.\ x\gamma_A \text{ is ground}) \longrightarrow (\forall x \in B.\ x\gamma_B \text{ is ground}) \longrightarrow (\forall x \in A.\ x\gamma = x\gamma_A) \wedge (\forall x \in B.\ x\gamma = x\gamma_B))$; an example of a function that fulfills this specification is $\lambda x.\ \mathsf{if}\,x \in A\,\mathsf{then}\,x\gamma_A\,\mathsf{else}\,x\gamma_B$. The length of a trail $\Gamma$ is denoted by $|\Gamma|$. The $n$th right-most element of a trail $\Gamma$ is denoted by $\Gamma[n]$; we use zero-based indexing where the right-most element is the 0th element. The Herbrand interpretation of a trail, denoted $\mathcal{HI}$, contains the atoms of all positive literals in the trail (i.e., $\mathcal{HI}\,\epsilon = \{\}$ and $\forall \Gamma\ A.\ \mathcal{HI}\,(\Gamma, A) = \{A\} \cup \mathcal{HI}\,\Gamma$ and $\forall \Gamma\ A.\ \mathcal{HI}\,(\Gamma, \neg A) = \mathcal{HI}\,\Gamma$).

The formalization also changes some existing concepts. No distinction is made between *atoms* and terms, so first-order terms are used everywhere in place of atoms. The level annotation of a *decision literal* is not required anymore and replaced by a † marker, it is now written $K^\dagger = \langle K; \dagger\rangle$ for some literal $K$. A *propagation literal* is written $(K\gamma_D)^{\langle K; D; \gamma_D\rangle} = \langle K\gamma_D; \langle D; K; \gamma_D\rangle\rangle$ for some literal $K$, clause $D$, and grounding $\gamma_D$. Note that the propagated literal is explicitly separated from its clause in the closure annotation; this eases the formulation of the additional invariants 5 and 6 of Lemma 5.1., that the respective clause is always false under the respective trail. For the *trail* $\Gamma, \mathcal{K}$, the Isabelle formalization uses the constructor $\mathsf{List.Cons}\,\mathcal{K}\,\Gamma$ which actually grows from right to left. However, we keep the well-established left-to-right convention in this chapter because it significantly eases the presentation. A state is now a tuple $\langle \Gamma; U; \mathcal{C}\rangle$ where $\Gamma$ is a trail, $U$ is a finite set of learned clauses, and $\mathcal{C}$ is an optional clause closure. The individual components can be selected with $\mathsf{trail}\,\langle \Gamma; U; \mathcal{C}\rangle = \Gamma$, $\mathsf{learned}\,\langle \Gamma; U; \mathcal{C}\rangle = U$, and $\mathsf{conflict}\,\langle \Gamma; U; \mathcal{C}\rangle = \mathcal{C}$. The *initial state* is $\langle \epsilon; \{\}; \dagger\rangle$, i.e., empty trail, no learned clauses, and no conflicting closure. The finite set of initial clauses $N$ and the bounding atom $\beta$ are no longer stored in the state but are rather parameters of the transition relation; this was done to highlight the fact that they are never modified by any rule. The natural number $k$ counting the number of decisions, used in Section 5.2 to determine an appropriate backtracking point, turned out not to be necessary and was dropped entirely. We assume the existence of a binary relation on atoms $\prec_B$ such that every atom has finitely many lesser atoms (i.e., $\forall \beta.\ \{t \mid t \prec_B \beta\}$ is finite) but dropped the requirement for $\prec_B$ to be a strict ordering total on ground terms. We also do not lift $\prec_B$ to literals and clauses, but always use it at the atom level. We define the relation $\preceq_B$ as the reflexive closure of $\prec_B$.

Remember from Chapter 3 that the function $\mathsf{gnd}_{\mathsf{cls}}$ denotes the set of all ground instances of a clause set. Its subset whose clauses are restricted to atoms less than or equal to a bound $\beta$ w.r.t. the strict ordering $\prec_B$ is defined such that $\forall N.\ \mathsf{gnd}_{\mathsf{cls}}^{\preceq_B \beta}\,N = \{C \in \mathsf{gnd}_{\mathsf{cls}}\,N \mid \forall L \in C.\ \mathsf{atom}\,L \preceq_B \beta\}$.

The transition relation $\Rightarrow_{\mathrm{SCL}}^{N,\beta}$ is a binary predicate between states and is parameterized by the finite set $N$ of initial clauses and the bounding atom $\beta$. It is defined as the disjunction of the following rules. Following each rule, we highlight the main differences from Section 5.2 not already covered.

**Propagate** $\langle \Gamma; U; \dagger\rangle \Rightarrow_{\mathrm{Propagate}}^{N,\beta} \langle \Gamma, (L\mu\gamma)^{\langle L\mu; C_0\mu;\gamma\rangle}; U; \dagger\rangle$
Side conditions:

1. $(L \vee C) \in N \cup U$
2. $\gamma$ is a grounding for $L \vee C$
3. $\forall K \in (L \vee C).\ \mathsf{atom}\,(K\gamma) \preceq_B \beta$
4. $C_0 = \{K \in C \mid K\gamma \neq L\gamma\}$
5. $C_1 = \{K \in C \mid K\gamma = L\gamma\}$
6. $C_0\gamma$ is false under $\Gamma$
7. $L\gamma$ is undefined in $\Gamma$
8. $\mu$ is an IMGU for $\{\mathsf{atom}\ K \mid K \in (L \vee C_1)\}$

Compared with Section 5.2, we express the splitting of $C$ into $C_0$ and $C_1$ formally as set operations and replace $\prec_B$ with $\preceq_B$. This replacement does not change the main characteristics of the calculus but allowing the bound $\beta$ to be in $\mathsf{gnd}_{\mathsf{cls}}^{\preceq_B\beta}\ N$ eases the proof of Lemma 5.23, where the largest element of the (finite) unsatisfiable core is directly used as new bound. There are also situations where the maximal element of a signature is required to derive a contradiction: a strict bound requires to artificially extend the signature while a nonstrict bound does not.

**Decide** $\langle \Gamma; U; \dagger\rangle \Rightarrow_{\mathrm{Decide}}^{N,\beta} \langle \Gamma, (L\gamma)^{\dagger}; U; \dagger\rangle$
Side conditions:

1. $(L \vee C) \in N$
2. $\gamma$ is a grounding for $L$
3. $L\gamma$ is undefined in $\Gamma$
4. $\mathsf{atom}\ L\gamma \preceq_B \beta$

Compared with Section 5.2, we replace $\prec_B$ with $\preceq_B$ and take the decision literal from $N$ instead of $N \cup U$. We proved that the ground instances of literals of $U$ are a subset of the ground instances of literals of $N$; it is thus redundant to also consider $U$ here.

**Conflict** $\langle \Gamma; U; \dagger\rangle \Rightarrow_{\mathrm{Conflict}}^{N,\beta} \langle \Gamma; U; \langle C; \gamma\rangle\rangle$
Side conditions:

1. $C \in N \cup U$
2. $\gamma$ is a grounding for $C$
3. $C\gamma$ is false under $\Gamma$

**Skip** $\langle \Gamma, \mathcal{K}; U; \langle C; \gamma\rangle\rangle \Rightarrow_{\mathrm{Skip}}^{N,\beta} \langle \Gamma; U; \langle C; \gamma\rangle\rangle$
Side conditions:

1. $\mathsf{comp}\,(\mathsf{lit}\,\mathcal{K}) \notin C\gamma$

**Factorize** $\langle \Gamma; U; \langle L' \vee L \vee C; \gamma\rangle\rangle \Rightarrow_{\mathrm{Factorize}}^{N,\beta} \langle \Gamma; U; \langle (L \vee C)\mu; \gamma\rangle\rangle$
Side conditions:

1. $L\gamma = L'\gamma$
2. $\mu$ is the IMGU for $\{\mathsf{atom}\ L, \mathsf{atom}\ L'\}$

**Resolve** $\langle \Gamma; U; \langle L \vee C; \gamma_C \rangle\rangle \Rightarrow^{N,\beta}_{\text{Resolve}} \langle \Gamma; U; \langle (C\rho_C \vee D\rho_D)\mu; \gamma \rangle\rangle$
Side conditions:

1. $\Gamma = \Gamma', (K\gamma_D)^{\langle K; D; \gamma_D \rangle}$
2. $K\gamma_D = \mathsf{comp}\ (L\gamma_C)$
3. $\rho_C$ and $\rho_D$ are renamings such that the variables of $(L \vee C)\rho_C$ and $(K \vee D)\rho_D$ are disjoint
4. $\mu$ is the IMGU for $\{\mathsf{atom}\ L\rho_C, \mathsf{atom}\ K\rho_D\}$
5. $\gamma'_C$ and $\gamma'_D$ are adaptations of $\gamma_C$ and $\gamma_D$ to the renamings $\rho_C$ and $\rho_D$ respectively
6. $\gamma$ is a merged grounding of $\gamma'_C$ for the variables of $(L \vee C)\rho_C$ and $\gamma'_D$ for the variables of $(K \vee D)\rho_D$

Note that the definition of IMGU and merged grounding imply the following equalities: $\mu \circledast \gamma = \gamma$, $L\rho_C\gamma = L\gamma_C$, $C\rho_C\gamma = C\gamma_C$, $K\rho_D\gamma = K\gamma_D$, and $D\rho_D\gamma = D\gamma_D$.

Compared with Section 5.2, we explicitly rename the merged clauses to avoid variable-name clashes instead of assuming disjoint variables, and use an abstract specification for the merged grounding instead of forcing substitution composition. The latter makes our rule more general by allowing more freedom to an implementation.

**Backtrack** $\langle \Gamma, \Gamma', K^\dagger; U; \langle L \vee C; \gamma \rangle\rangle \Rightarrow^{N,\beta}_{\text{Backtrack}} \langle \Gamma; \{L \vee C\} \cup U; \dagger \rangle$
Side conditions:

1. $K = \mathsf{comp}\ (L\gamma)$
2. $\nexists\gamma'.\ (L \vee C)\gamma'$ is ground and false under $\Gamma$

Note that backtracking to an empty trail, i.e., with $\Gamma = \epsilon$ is always possible because a nonempty clause cannot be false under $\epsilon$. In practice, one wants to backtrack as little as possible to reuse as much of the trail as possible.

Compared with Section 5.2, we allow backtracking to any nonconflicting trail instead of specifying the position. This makes our rule more general by, again, allowing more freedom to an implementation. The minimally backtracking strategy introduced in Definition 5.4 brings back equivalence to the Backtrack rule of Section 5.2.

### Isabelle Technicalities

We define the SCL rules in the scl_fol_calculus locale. It fixes an abstract binary relation $\prec_B$ as a locale parameter and assumes that it bounds a finite number of atoms. It also fixes an abstract function to generate variable renamings as a locale parameter and assumes its correctness; this function is not required for the specification of the calculus but is required in multiple proofs. We provided an example instantiation of all locale assumptions to ensure that they can be fulfilled and are not contradictory. Most of the following definitions and theorems are in the context of this locale. Each SCL rule is defined separately as an inductive predicate. Having separate definitions allows to refer to the rules individually in subsequent definitions and theorems. Using inductive predicates, as opposed to plain definitions,

is convenient because Isabelle automatically generates some useful introduction and elimination lemmas, and configures structured Isar syntax for case analysis.

From the SCL rules, we can prove a number of invariants about states. Most of them are intuitive while few are technicalities of the Isabelle formalization. We will use the invariants as hypotheses for several of the main lemmas and theorems.

**Lemma 5.1** (Invariants)**.** *Let $\langle \Gamma; U; \mathcal{C}\rangle$ be a state w.r.t. $\Rightarrow_{SCL}^{N,\beta}$. The following invariants hold for the initial state $\langle \epsilon; \{\}; \dagger\rangle$ and are each individually preserved by the SCL rules.*

1. *All annotated literals in $\Gamma$ are ground:*
   - $\forall K \in \{\mathsf{lit}\ \mathcal{K} \mid \mathcal{K} \in \mathsf{set}\ \Gamma\}.\ K$ *is a ground literal*
2. *The atoms of all annotated literals in $\Gamma$ are $\preceq_B \beta$:*
   - $\forall K \in \{\mathsf{lit}\ \mathcal{K} \mid \mathcal{K} \in \mathsf{set}\ \Gamma\}.\ \mathsf{atom}\ K \preceq_B \beta$
3. *All annotated literals in $\Gamma$ are undefined in their respective subtrail of $\Gamma$:*
   - $\forall \Gamma'\ \mathcal{K}\ \Gamma''.\ \Gamma = \Gamma', \mathcal{K}, \Gamma'' \longrightarrow \mathsf{lit}\ \mathcal{K}$ *is undefined in* $\Gamma'$
4. *All closures in $\Gamma$ and $\mathcal{C}$ are ground:*
   - $\forall \mathcal{K} \in \mathsf{set}\ \Gamma.\ \forall D\ K\ \gamma.\ \mathcal{K} = (K\gamma)^{\langle K; D; \gamma\rangle} \longrightarrow D\gamma$ *is ground*
   - $\forall C\ \gamma.\ \mathcal{C} = \langle C; \gamma\rangle \longrightarrow C\gamma$ *is ground*
5. *All closures in $\Gamma$ and $\mathcal{C}$ are false under their respective subtrail of $\Gamma$:*
   - *invariant 4 holds*
   - $\forall D\ K\ \gamma\ \Gamma'\ \Gamma''.\ \Gamma = \Gamma', (K\gamma)^{\langle K; D; \gamma\rangle}, \Gamma'' \longrightarrow D\gamma$ *is false under* $\Gamma'$
   - $\forall C\ \gamma.\ \mathcal{C} = \langle C; \gamma\rangle \longrightarrow C\gamma$ *is false under* $\Gamma$
6. *All propagated literals in $\Gamma$ are ground instances of the literal in their closure annotations:*
   - $\forall \mathcal{K} \in \mathsf{set}\ \Gamma.\ \forall D\ K\ \gamma.\ \mathsf{ann}\ \mathcal{K} = \langle D; K; \gamma\rangle \longrightarrow \mathsf{lit}\ \mathcal{K} = K\gamma$
7. *The complements of all propagated literals in $\Gamma$ are absent from their closure annotation:*
   - $\forall \mathcal{K} \in \mathsf{set}\ \Gamma.\ \forall D\ K\ \gamma.\ \mathcal{K} = (K\gamma)^{\langle K; D; \gamma\rangle} \longrightarrow \mathsf{comp}\ (K\gamma) \notin D\gamma$
8. *All literals of the clauses in $\Gamma$'s propagating clauses, $U$, and $\mathcal{C}$ have a corresponding, more general literal in $N$:*
   - $\forall D \in \{D \mid (K\gamma)^{\langle K; D; \gamma\rangle} \in \mathsf{set}\ \Gamma\} \cup U \cup (\mathsf{case}\ \mathcal{C}\ \mathsf{of}\ \dagger \Rightarrow \{\} \mid \langle C; \gamma\rangle \Rightarrow \{C\}).$ $\forall K \in D.\ \exists D' \in N.\ \exists K' \in D'.\ \exists \sigma.\ K'\sigma = K$
9. *All annotated literals in $\Gamma$ have a corresponding more general literal either in $N$ or in $U$:*
   - $\forall \mathcal{K} \in \mathsf{set}\ \Gamma.\ \exists C \in N \cup U.\ \exists L \in C.\ \exists \sigma.\ L\sigma = \mathsf{lit}\ \mathcal{K}$
10. *All clauses in $\Gamma$, $U$, and $\mathcal{C}$ are entailed by $N$:*
    - $\forall \mathcal{K} \in \mathsf{set}\ \Gamma.\ \forall D\ K\ \gamma.\ \mathcal{K} = (K\gamma)^{\langle K; D; \gamma\rangle} \longrightarrow N \models \{K \vee D\}$
    - $N \models U$
    - $\forall C\ \gamma.\ \mathcal{C} = \langle C; \gamma\rangle \longrightarrow N \models \{C\}$

The SCL calculus is defined as a transition system where many decisions are deferred to strategies. A *strategy* specifies a transition system whose transitions are a

subset of those from an existing transition system. We say that a strategy $\mathcal{S}$ *restricts* a transition system $\mathcal{T}$ (or symmetrically that $\mathcal{T}$ is *restricted* by $\mathcal{S}$) if every valid transition of $\mathcal{S}$ is also a valid transition of $\mathcal{T}$ (i.e., $\forall x\ y.\ \mathcal{S}\ x\ y \longrightarrow \mathcal{T}\ x\ y$). Note that strategies can be chained to iteratively apply more restrictions.

We define the reasonable and regular strategies restricting the $\Rightarrow_{\text{SCL}}^{N,\beta}$ relation to prove the main results of this chapter.

**Definition 5.2.** *Let $N$ be a final set of initial clauses. Let $\beta$ be a bound. The reasonable strategy $\Rightarrow_{\textit{Rea-SCL}}^{N,\beta}$ restricts the SCL calculus by preventing decisions that immediately lead to a conflict (such situations could be replaced by a propagation):*

$$\forall S\ S'.\ S \Rightarrow_{\textit{Rea-SCL}}^{N,\beta} S' \longleftrightarrow$$
$$S \Rightarrow_{\textit{SCL}}^{N,\beta} S' \wedge (S \Rightarrow_{\textit{Decide}}^{N,\beta} S' \longrightarrow (\nexists S''.\ S' \Rightarrow_{\textit{Conflict}}^{N,\beta} S''))$$

**Definition 5.3.** *Let $N$ be a final set of initial clauses. Let $\beta$ be a bound. The regular strategy $\Rightarrow_{\textit{Reg-SCL}}^{N,\beta}$ restricts the reasonable strategy by prioritizing the conflict rule to any other:*

$$\forall S\ S'.\ S \Rightarrow_{\textit{Reg-SCL}}^{N,\beta} S' \longleftrightarrow$$
$$S \Rightarrow_{\textit{Rea-SCL}}^{N,\beta} S' \wedge ((\exists S''.\ S \Rightarrow_{\textit{Conflict}}^{N,\beta} S'') \longrightarrow S \Rightarrow_{\textit{Conflict}}^{N,\beta} S')$$

While not required for the coming nonredundancy and termination results, we also define the minimally backtracking strategy to express the constraint on the backtracking position found in Section 5.2.

**Definition 5.4.** *Let $N$ be a final set of initial clauses. Let $\beta$ be a bound. The minimally backtracking strategy $\Rightarrow_{\textit{Min-Bac-SCL}}^{N,\beta}$ restricts the regular strategy by requiring that backtracking removes the shortest possible suffix of the trail:*

$$\forall S\ S'.\ S \Rightarrow_{\textit{Min-Bac-SCL}}^{N,\beta} S' \longleftrightarrow$$
$$S \Rightarrow_{\textit{Reg-SCL}}^{N,\beta} S' \wedge (S \Rightarrow_{\textit{Backtrack}}^{N,\beta} S' \longrightarrow$$
$$\mathsf{trail}\ S' \textit{ is the longest prefix of } \mathsf{trail}\ S$$
$$\textit{not in conflict with the learned clause})$$

All three strategies build on one-another and ultimately restrict the SCL calculus. We can express this formally as implications, of which the first can be used to show that coming results (e.g., Corollaries 5.13 and 5.21) also hold for the minimally backtracking strategy.

**Lemma 5.5.** *The minimally backtracking strategy restricts the regular strategy, which restricts the reasonable strategy, which restricts the SCL calculus:*

- $\forall N\ \beta\ S\ S'.\ S \Rightarrow_{\textit{Min-Bac-SCL}}^{N,\beta} S' \longrightarrow S \Rightarrow_{\textit{Reg-SCL}}^{N,\beta} S'$
- $\forall N\ \beta\ S\ S'.\ S \Rightarrow_{\textit{Reg-SCL}}^{N,\beta} S' \longrightarrow S \Rightarrow_{\textit{Rea-SCL}}^{N,\beta} S'$
- $\forall N\ \beta\ S\ S'.\ S \Rightarrow_{\textit{Rea-SCL}}^{N,\beta} S' \longrightarrow S \Rightarrow_{\textit{SCL}}^{N,\beta} S'$

The bounding atom $\beta$ restricts the calculus to only consider the finitely many ground atoms less than or equal to $\beta$ w.r.t. $\prec_B$; this will play an important role in the termination proof. When SCL terminates, it either derived a contradiction, or it found a model for the bounded ground instances of the initial clauses. Because $\beta$ is usually chosen heuristically, the model might be unsatisfactory for the considered use case and one may want to continue execution with a bigger bound. This is allowed if the new bound properly extends the previous bound $\beta$ w.r.t. $\preceq_B$.

**Theorem 5.6** (Monotonicity w.r.t. Bound)**.** *Let $\beta$ and $\beta'$ be bounds. If the ground atoms bound by $\beta$ are a subset of the ground atoms bound by $\beta'$ (i.e., $\forall A.\ A$ is ground $\longrightarrow A \preceq_B \beta \longrightarrow A \preceq_B \beta'$), then the SCL, reasonable SCL, regular SCL, and minimally backtracking SCL transitions w.r.t. $\beta$ are also transitions w.r.t. $\beta'$:*

- $\forall N\ S\ S'.\ S \Rightarrow_{SCL}^{N,\beta} S' \longrightarrow S \Rightarrow_{SCL}^{N,\beta'} S'$
- $\forall N\ S\ S'.\ S \Rightarrow_{Rea\text{-}SCL}^{N,\beta} S' \longrightarrow S \Rightarrow_{Rea\text{-}SCL}^{N,\beta'} S'$
- $\forall N\ S\ S'.\ S \Rightarrow_{Reg\text{-}SCL}^{N,\beta} S' \longrightarrow S \Rightarrow_{Reg\text{-}SCL}^{N,\beta'} S'$
- $\forall N\ S\ S'.\ S \Rightarrow_{Min\text{-}Bac\text{-}SCL}^{N,\beta} S' \longrightarrow S \Rightarrow_{Min\text{-}Bac\text{-}SCL}^{N,\beta'} S'$

Theorem 5.6 implies that all properties w.r.t. a bound $\beta$ also hold w.r.t. a compatible bound $\beta'$. Its hypothesis is fulfilled if $\preceq_B$ is transitive on ground atoms, $\beta$ and $\beta'$ are ground atoms, and $\beta \preceq_B \beta'$. The bounding atom could even be increased at any point in an SCL run, not just when the calculus terminated.

The different rules and strategies considered so far express a single step of computation for the SCL calculus; they offer a good level of granularity to both understand and mechanize the details of the calculus. But several results of the following sections ought to express properties of the calculus as a whole. We express such results in terms of a run from the initial state. A *run* is the reflexive, transitive closure of a rule or strategy (e.g., $S\ (\Rightarrow_{\text{SCL}}^{N,\beta})^*\ S'$ is an SCL run from the state $S$ to the state $S'$).

### 5.3.1 Soundness

The soundness of the individual SCL rules is shown by invariant 10. We now consider the soundness of terminating runs of the SCL calculus as a whole.

**Theorem 5.7** (Correct Termination)**.** *Let $S = \langle \Gamma; U; \mathcal{C} \rangle$ be a state w.r.t. $\Rightarrow_{SCL}^{N,\beta}$. If invariants 2, 3, 5, 6 and 10 hold for $S$, and if $S$ is a stuck state with some restrictions, formally if all of the following hold*

- $\nexists S'.\ S \Rightarrow_{Propagate}^{N,\beta} S'$,
- $\nexists S'.\ S \Rightarrow_{Decide}^{N,\beta} S' \wedge (\nexists S''.\ S' \Rightarrow_{Conflict}^{N,\beta} S'')$,
- $\nexists S'.\ S \Rightarrow_{Conflict}^{N,\beta} S'$,
- $\nexists S'.\ S \Rightarrow_{Skip}^{N,\beta} S'$,
- $\nexists S'.\ S \Rightarrow_{Resolve}^{N,\beta} S'$,
- $\nexists S'.\ S \Rightarrow_{Backtrack}^{N,\beta} S'$ *and the backtracking is minimal,*

*then either the conflicting clause clause is $\bot$ and the set of ground instances of the initial clauses is unsatisfiable (i.e., $(\exists \gamma.\ \mathcal{C} = \langle \bot; \gamma \rangle) \wedge (\nexists \mathcal{I}.\ \mathcal{I} \models_{\mathsf{cls}} (\mathsf{gnd}_{\mathsf{cls}}\ N))$), or there is no conflicting clause and the set of ground instances of the initial clauses with atoms less than or equal to the bound is satisfiable by the trail (i.e., $\mathcal{C} = \dagger \wedge \mathcal{HI}\ \Gamma \models_{\mathsf{cls}} (\mathsf{gnd}_{\mathsf{cls}}^{\preceq_B \beta}\ N)$).*

Note that no hypothesis restricts the usage of the Factorize rule because it is an optional step of conflict resolution that has no impact on satisfiability.

Theorem 5.7 holds for a family of strategies, in contrast to Theorem 5 from Bromberger et al., which was only shown for what is here called the minimally

backtracking strategy. This family of strategies contains any strategy that preserves the required invariants and is restricted by the minimally backtracking strategy. From Lemma 5.5 we know that these two requirements are fulfilled by the SCL relation but also by the reasonable, regular, and minimally backtracking strategies. This leads to a more intuitive corollary based on runs.

**Corollary 5.8.** *If an SCL, a reasonable SCL, a regular SCL, or a minimally backtracking SCL run starting from the initial state* $\langle \epsilon; \{\}; \dagger \rangle$ *terminates in a state* $S = \langle \Gamma; U; \mathcal{C} \rangle$ *w.r.t.* $\Rightarrow_{SCL}^{N,\beta}$*, formally if any of the following holds*

- $\langle \epsilon; \{\}; \dagger \rangle \, (\Rightarrow_{SCL}^{N,\beta})^* \, S \wedge (\nexists S'.\ S \Rightarrow_{SCL}^{N,\beta} S')$,
- $\langle \epsilon; \{\}; \dagger \rangle \, (\Rightarrow_{Rea\text{-}SCL}^{N,\beta})^* \, S \wedge (\nexists S'.\ S \Rightarrow_{Rea\text{-}SCL}^{N,\beta} S')$,
- $\langle \epsilon; \{\}; \dagger \rangle \, (\Rightarrow_{Reg\text{-}SCL}^{N,\beta})^* \, S \wedge (\nexists S'.\ S \Rightarrow_{Reg\text{-}SCL}^{N,\beta} S')$, *or*
- $\langle \epsilon; \{\}; \dagger \rangle \, (\Rightarrow_{Min\text{-}Bac\text{-}SCL}^{N,\beta})^* \, S \wedge (\nexists S'.\ S \Rightarrow_{Min\text{-}Bac\text{-}SCL}^{N,\beta} S')$,

*then the conclusion of Theorem 5.7 holds.*

Note that each strategy is used with positive polarity in the "run" hypothesis and negative polarity in the "no-more-step" hypothesis. For this reason, it is impossible to provide a corollary with a single requirement to restrict or be restricted by any known strategy.

### 5.3.2 Nonredundancy of Learned Clauses

Traditional saturation-based calculi for first-order logic (e.g., resolution and superposition) can learn redundant clauses and thus their implementations require costly checks for nonredundancy. SCL(FOL) learns only nonredundant clauses. Thus, an implementation would not need to check for (forward) nonredundancy. We first repeat the definition of *standard redundancy* as used in the saturation framework [119].

**Definition 5.9** (Redundant Clause)**.** *Let* $N$ *be a finite clause set. Let* $C$ *be a clause. Let* $\prec$ *be a strict ordering on clauses. We say that* $D$ *is* redundant *w.r.t.* $N$ *and* $\prec$ *if all its ground instances are entailed by the lesser ground instances of* $N$ *(i.e.,* $\forall D' \in \mathsf{gnd}_{\mathsf{cls}}\ D.\ \{C' \in \mathsf{gnd}_{\mathsf{clss}}\ N \mid C' \prec D'\} \models \{D'\}$*).*

We first prove nonredundancy of learned clauses w.r.t. a trail-induced dynamic ordering and then lift this result to nonredundancy w.r.t. a static ordering.

**Definition 5.10** (Trail-Induced Atom Ordering)**.** *Let* $\Gamma$ *be a trail of annotated literal.* $\Gamma$ *induces a well-founded, strict partial ordering* $\prec^\Gamma$*, total on all atoms in* $\Gamma$*'s literals. Consider the case when* $\Gamma$ *has the form* $L_n{}^{ann_n}, \ldots, L_2{}^{ann_2}, L_1{}^{ann_1}, L_0{}^{ann_0}$*, we have the following ordering.*

$$\mathsf{atom}\ L_n \prec^\Gamma \cdots \prec^\Gamma \mathsf{atom}\ L_2 \prec^\Gamma \mathsf{atom}\ L_1 \prec^\Gamma \mathsf{atom}\ L_0$$

*In other words, "older" elements on the left are smaller than "newer" elements on the right. We specify* $\prec^\Gamma$ *formally with the following equivalence:*

$$\begin{aligned} &\forall t_1\ t_2.\ t_1 \prec^\Gamma t_2 \longleftrightarrow \\ &\quad (\exists i < |\Gamma|.\ \exists j < i.\ t_1 = \mathsf{atom}\ (\mathsf{lit}\ (\Gamma[i])) \wedge t_2 = \mathsf{atom}\ (\mathsf{lit}\ (\Gamma[j]))) \end{aligned}$$

Compared with Bromberger et al., the trail-induced ordering is defined on atoms instead of literals and we prove nonredundancy for any lifting to literals.

**Theorem 5.11** (Dynamic Nonredundancy of Learned Clauses)**.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. Following conflict resolution in a regular run, formally if the following conditions hold,*

- $\langle \epsilon; \{\}; \dagger\rangle\ (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^*\ \langle \Gamma; U; \dagger\rangle$,
- $\langle \Gamma; U; \dagger\rangle\ \Rightarrow^{N,\beta}_{\textit{Conflict}}\ S_1$,
- $S_1\ (\Rightarrow^{N,\beta}_{\textit{Skip,Factorize,Resolve}})^+\ S_n$, *and*
- $S_n\ \Rightarrow^{N,\beta}_{\textit{Backtrack}}\ S_{1+n}$,

*then neither is the learned clause generalized by any known clause (i.e., $\nexists D \in N \cup U.\ \exists\sigma.\ D\sigma = \mathsf{conflict}\ S_n$), nor is it redundant w.r.t. $N \cup U$ and the ordering we get by first lifting the trail-induced ordering $\prec^\Gamma$ from atoms to literals and then taking its multiset extension.*

Dynamic nonredundancy with respect to the trail-induced ordering does not by itself release an implementation from performing backward nonredundancy checks, but it is a strong guarantee on the quality of learned clauses. For backward redundancy checks, an ordering needs to be used that encompasses all dynamic trail-induced orders. An ordering based on a strict multiset relation has this property. So for backward redundancy we can, e.g., delete subsumed clauses.

**Corollary 5.12** (Static Nonsubsumption of Learned Clauses)**.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. If a regular run starting from the initial state $\langle \epsilon; \{\}; \dagger\rangle$ learns a clause $C$, formally if the following conditions hold,*

- $\langle \epsilon; \{\}; \dagger\rangle\ (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^*\ \langle \Gamma; U; \langle C; \gamma\rangle\rangle$ *and*
- $\langle \Gamma; U; \langle C; \gamma\rangle\rangle\ \Rightarrow^{N,\beta}_{\textit{Backtrack}}\ S$,

*then $C$ is not subsumed by any initial or learned clause (i.e., $\nexists D \in N \cup U.\ \exists\sigma.\ D\sigma \subseteq C$).*

All nonredundancy results can be generalized to an arbitrary strategy restricting the regular strategy. We only show one example here and refer the reader to the formalization for the others.

**Corollary 5.13.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. Let $\mathcal{R}$ be the transition relation of some strategy. Following conflict resolution in the run of a strategy restricting regular SCL, formally if the following conditions hold,*

- $\langle \epsilon; \{\}; \dagger\rangle\ (\Rightarrow^{N,\beta}_{\textit{Strategy}})^*\ \langle \Gamma; U; \dagger\rangle$,
- $\langle \Gamma; U; \dagger\rangle\ \Rightarrow^{N,\beta}_{\textit{Conflict}}\ S_1$,
- $S_1\ (\Rightarrow^{N,\beta}_{\textit{Skip,Factorize,Resolve}})^+\ S_n$,
- $S_n\ \Rightarrow^{N,\beta}_{\textit{Backtrack}}\ S_{1+n}$, *and*
- $\forall S\ S'.\ \mathcal{R}\ S\ S' \longrightarrow S\ \Rightarrow^{N,\beta}_{\textit{Reg-SCL}}\ S'$,

*then neither is the learned clause generalized by any initial or learned clause (i.e., $\nexists D \in N \cup U.\ \exists\sigma.\ D\sigma = \mathsf{conflict}\, S_n$), nor is it redundant w.r.t. $N \cup U$ and the ordering we get by first lifting the trail-induced ordering $\prec^\Gamma$ from atoms to literals and then taking its multiset extension.*

During the development of this formalization, we discovered that the original Backtrack rule found in [26] allows to learn a duplicate of the last learned clause, which violates the stated nonredundancy of learned clauses. The original Backtrack rule ensures that the conflict closure is not false under the new trail, but the learned clause could still be in conflict w.r.t. another grounding. Following this conflict, the Backtrack rules would be immediately applicable and would learn the same clause again. This could only happen a finite number of times as backtracking reduces the length of the (finite) trail.

**Example 5.14.** *Let* $\mathsf{P}$, $\mathsf{Q}$, $\mathsf{R}$, *and* $\mathsf{S}$ *be predicate symbols. Let* $v$, $w$, $x$, $y$, *and* $z$ *be variable symbols. Let* $\mathsf{a}$ *and* $\mathsf{b}$ *be ground terms. Consider the clause set* $N = \{\mathsf{P}(x), \mathsf{Q}(y), \neg\mathsf{Q}(z) \vee \mathsf{R}(z), \neg\mathsf{R}(w) \vee \mathsf{S}(w), \neg\mathsf{P}(v) \vee \neg\mathsf{S}(v)\}$ *and a big enough* $\beta$. *The following SCL run was valid with the original Backtrack rule. Note that the notation for the trail was shortened to save space.*

$$
\begin{aligned}
& \langle \epsilon; \{\}; \dagger \rangle \\
(\Rightarrow^{N,\beta}_{Decide})^* \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}), \mathsf{Q}(\mathsf{b}); \{\}; \dagger \rangle \\
(\Rightarrow^{N,\beta}_{Propagate})^* \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}), \mathsf{Q}(\mathsf{b}), \mathsf{R}(\mathsf{b})^{\langle \mathsf{R}(z); \neg\mathsf{Q}(z); z \mapsto \mathsf{b} \rangle}, \mathsf{S}(\mathsf{b})^{\langle \mathsf{S}(w); \neg\mathsf{R}(w); w \mapsto \mathsf{b} \rangle}; \{\}; \dagger \rangle \\
\Rightarrow^{N,\beta}_{Conflict} \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}), \mathsf{Q}(\mathsf{b}), \mathsf{R}(\mathsf{b})^{\langle \mathsf{R}(z); \neg\mathsf{Q}(z); z \mapsto \mathsf{b} \rangle}, \mathsf{S}(\mathsf{b})^{\langle \mathsf{S}(w); \neg\mathsf{R}(w); w \mapsto \mathsf{b} \rangle}; \{\}; \langle \neg\mathsf{P}(v) \vee \neg\mathsf{S}(v); v \mapsto \mathsf{b} \rangle \rangle \\
\Rightarrow^{N,\beta}_{Resolve+Skip} \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}), \mathsf{Q}(\mathsf{b}), \mathsf{R}(\mathsf{b})^{\langle \mathsf{R}(z); \neg\mathsf{Q}(z); z \mapsto \mathsf{b} \rangle}; \{\}; \langle \neg\mathsf{P}(v) \vee \neg\mathsf{R}(v); v \mapsto \mathsf{b} \rangle \rangle \\
\Rightarrow^{N,\beta}_{Resolve+Skip} \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}), \mathsf{Q}(\mathsf{b}); \{\}; \langle \neg\mathsf{P}(v) \vee \neg\mathsf{Q}(v); v \mapsto \mathsf{b} \rangle \rangle \\
\Rightarrow^{N,\beta}_{Backtrack} \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}), \mathsf{P}(\mathsf{b}); \{\neg\mathsf{P}(v) \vee \neg\mathsf{Q}(v)\}; \dagger \rangle \\
\Rightarrow^{N,\beta}_{Conflict+Skip} \quad & \langle \mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{a}); \{\neg\mathsf{P}(v) \vee \neg\mathsf{Q}(v)\}; \langle \neg\mathsf{P}(v) \vee \neg\mathsf{Q}(v); v \mapsto \mathsf{a} \rangle \rangle \\
\Rightarrow^{N,\beta}_{Backtrack} \quad & \langle \mathsf{P}(\mathsf{a}); \{\neg\mathsf{P}(v) \vee \neg\mathsf{Q}(v)\}; \dagger \rangle
\end{aligned}
$$

This counterexample was only discovered when we failed to prove Theorem 5.11 in Isabelle. Note that this formalization is based on an early version of Bromberger et al.'s paper and was developed in parallel to the final version, which originally inherited the Backtrack rule from [52].

The solution, which was promptly integrated into this formalization and Bromberger et al., is for the Backtrack rule to find a position without conflict w.r.t. the learned clause. Note that the original Backtrack rule reaches such a state after having learned the same clause finitely often, which has no effect on the set of learned clauses because sets ignore duplicates. Thus, the original Backtrack rule did not invalidate the other properties of the SCL calculus. This discovery is strong evidence of the usefulness of mechanized formalization for both published work and ongoing research: the Isabelle formalization lead to the discovery of a previously unknown bug and it guided the development of the refinement.

### 5.3.3 Termination

A calculus expressed as a state machine terminates if the transition relation starting from the initial state is well-founded following the arrow direction. We prove well-foundedness of regular SCL in three steps:

1. We first prove well-foundedness of SCL without backtracking, denoted $\Rightarrow^{N,\beta}_{\text{SCL-no-Back}}$.
2. We then prove that a regular run can only learn finitely many clauses.
3. From these two results we finally prove well-foundedness of regular SCL.

Step 1 is novel to the formalization. Prior work by Bromberger et al. focuses exclusively on the Backtrack rule (step 2) to prove termination of regular SCL (step 3). Also novel to the formalization are decreasing measuring functions for steps 1 and 2.

**Definition 5.15.** *The function $\mathcal{M}_3$ provides a measure that decreases at each step of SCL without backtracking. It maps a bounding atom and a state to a 4-tuple. The tuple elements are (1) a Boolean identifying whether the state is conflict-free, (2) a (finite) set overapproximating the literals that could be added to the trail, (3) a (finite) list overapproximating the numbers of resolution steps that could be performed at each position in the trail, and (4) the (finite) cardinality of the conflicting clause.*

$$\forall \beta\ \Gamma.\ \mathcal{M}_1\ \beta\ \Gamma\ =\ \{L \mid \mathsf{atom}\ L \preceq_B \beta\} - \{\mathsf{lit}\ \mathcal{K} \mid \mathcal{K} \in \mathsf{set}\ \Gamma\}$$

$$\begin{aligned}
\forall C.\ \mathcal{M}_2\ \epsilon\ C\ &= \epsilon \\
\forall \Gamma\ K\ C.\ \mathcal{M}_2\ (\Gamma, K^\dagger)\ C\ &= \mathcal{M}_2\ \Gamma\ C, 0 \\
\forall \Gamma\ K\ D\ \gamma\ C.\ \mathcal{M}_2\ (\Gamma, (K\gamma)^{\langle K; D; \gamma\rangle})\ C\ &= (\mathsf{let}\ n = \mathsf{count}\ C\ (\mathsf{comp}\ (K\gamma))\ \mathsf{in} \\
&\quad\ \mathcal{M}_2\ \Gamma\ C \vee \mathsf{repeat}\ n\ (D\gamma), n)
\end{aligned}$$

$$\begin{aligned}
\forall \beta\ \Gamma\ U.\ \mathcal{M}_3\ \beta\ \langle \Gamma; U; \dagger\rangle\ &= \langle \mathsf{True}; \mathcal{M}_1\ \beta\ \Gamma; \epsilon; 0\rangle \\
\forall \beta\ \Gamma\ U\ C\ \gamma.\ \mathcal{M}_3\ \beta\ \langle \Gamma; U; \langle C; \gamma\rangle\rangle\ &= \langle \mathsf{False}; \{\}; \mathcal{M}_2\ \Gamma\ C; |C|\rangle
\end{aligned}$$

Using $\mathcal{M}_3$, we can prove termination of SCL without backtracking (step 1).

**Theorem 5.16** (Termination of SCL without Backtracking)**.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. SCL without backtracking is well-founded on all states reachable by a run starting from the initial state (i.e., on $\{S \mid \langle \epsilon; \{\}; \dagger\rangle\ (\Rightarrow^{N,\beta}_{\text{SCL-no-Back}})^*\ S\}$).*

**Remark 5.17.** *The well-founded predicate* wfp *$R$, found in the Isabelle distribution, expresses that a binary relation $R$ is well-founded on all inputs. However, SCL without backtracking does not terminate on all inputs but only on those that satisfy some specific invariants enforced by the run starting from the initial state.*

*One possible solution would have been to restrict the relation to only those arguments that fulfill the invariants. But instead, we defined a new predicate* wfp_on *$\mathcal{X}$ $R$ that expresses well-foundedness of $R$ only w.r.t. a set $\mathcal{X}$ of elements. The two methods are equivalent, but the restricted predicate has the advantage of more clearly distinguishing the two concepts of relation and input values. It also follows the convention used by many other predicates in the Isabelle distribution.*

*We integrated our new predicate* wfp_on *into the Isabelle distribution and redefined* wfp *in terms of it.*

We now turn to proving termination of regular SCL with backtracking by first defining an appropriate measuring function.

**Definition 5.18.** *The function $\mathcal{M}_4$ provides a measure that decreases at each step of the rule Backtrack. It maps a bounding atom and a state to a finite set of clauses without duplicated literals. It computes an overapproximation of the set of clauses that could potentially be learned modulo duplicate literals.*

$$\forall \beta\ S.\ \mathcal{M}_4\ \beta\ S\ =\ 2^{\{L \mid \mathsf{atom}\ L \preceq_B \beta\}} - \{\mathsf{set}\ C \mid C \in \mathsf{gnd}_{\mathsf{cls}}\ (\mathsf{learned}\ S)\}$$

We then prove that $\mathcal{M}_4$ decreases every time we learn a new clause (step 2).

**Lemma 5.19.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. Following conflict resolution in a regular run, formally if the following conditions hold,*

- $\langle \epsilon; \{\}; \dagger\rangle \, (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^* \, \langle \Gamma; U; \dagger\rangle$,
- $\langle \Gamma; U; \dagger\rangle \Rightarrow^{N,\beta}_{\textit{Conflict}} S_1$,
- $S_1 \, (\Rightarrow^{N,\beta}_{\textit{Skip,Factorize,Resolve}})^+ \, S_n$, *and*
- $S_n \Rightarrow^{N,\beta}_{\textit{Backtrack}} S_{1+n}$,

*then*

1. *the ground instance of the learned clause is distinct from all ground instances of initial and learned clauses modulo duplicate literals (i.e., $\exists C\, \gamma.\ \mathsf{conflict}\, S_n = \langle C; \gamma\rangle \wedge \mathsf{set}\, (C\gamma) \notin \{\mathsf{set}\, D \mid D \in \mathsf{gnd}_{\mathsf{cls}}\, (N \cup U)\}$), and*
2. *the set of clauses that could potentially be learned modulo duplicate literals strictly diminishes (i.e., $\mathcal{M}_4\, \beta\, S_{1+n} \subset \mathcal{M}_4\, \beta\, S_n$).*

Lemma 5.19 is novel to the formalization. Together with Theorem 5.16 it allows us to prove termination of regular SCL with backtracking (step 3).

**Theorem 5.20** (Termination of Regular SCL)**.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. Regular SCL is well-founded on all states reachable by a run starting from the initial state (i.e., on $\{S \mid \langle \epsilon; \{\}; \dagger\rangle \, (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^* \, S\}$).*

All termination results can be generalized to an arbitrary strategy restricting the regular strategy. We only show one example here and refer the reader to the formalization for the others.

**Corollary 5.21.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. Let $\mathcal{R}$ be the transition relation of some strategy. If $\mathcal{R}$ restricts regular SCL (i.e., $\forall S\, S'.\ \mathcal{R}\, S\, S' \longrightarrow S \Rightarrow^{N,\beta}_{\textit{Reg-SCL}} S'$), then it is well-founded on all states reachable by a run starting from the initial state (i.e., on $\{S \mid \mathcal{R}\, \langle \epsilon; \{\}; \dagger\rangle\, S\}$).*

All theorems until now were first expressed and proven using invariants and then the versions expressed using runs were derived. However, Theorem 5.20 posed an interesting problem because its proof requires the backtracking step to have knowledge of the trail when a conflict last occurred. But this information is lost in the SCL state due to the Skip rule shrinking the trail. We did define an invariant that expresses the historical form of the trail and its properties derived from the regular strategy, but it is complex and the added value compared with working directly on a regular run is questionable. We refer the interested reader to the lemma termination_regular_scl_invars of the Isabelle/HOL formalization for more details.

Together, soundness and termination allow us to prove refutational completeness of the regular SCL calculus w.r.t. a fixed bound.

**Theorem 5.22** (Completeness w.r.t. Bound)**.** *Let $N$ be a finite clause set. Let $\beta$ be a bound. If the ground instances of the initial clauses with atoms less than or equal to $\beta$ are unsatisfiable (i.e., $\nexists \mathcal{I}.\ \mathcal{I} \models_{\mathsf{cls}} (\mathsf{gnd}^{\preceq_B \beta}_{\mathsf{cls}}\, N)$), then all regular SCL runs starting from the initial state terminate and derive the conflicting clause $\bot$. In other words,*

1. *there is no infinite regular run starting from the initial state, and*

2. $\forall S.\ \langle \epsilon; \{\}; \dagger\rangle\ (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^*\ S \longrightarrow (\nexists S'.\ S \Rightarrow^{N,\beta}_{\textit{Reg-SCL}} S') \longrightarrow (\exists \gamma.\ \mathsf{conflict}\ S = \langle \bot; \gamma\rangle)$.

Theorem 5.22 is only defined w.r.t. a bound, but fortunately we can prove that there must always exist an appropriate bound.

**Lemma 5.23.** *Let $N$ be a finite clause set. If the relation $\prec_B$ is a well-founded, strict ordering, total on ground atoms and the ground instances of the initial clauses are unsatisfiable (i.e., $\nexists \mathcal{I}.\ \mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}_{\mathsf{clss}}\ N)$), then there exists a bound such that the ground instances of the initial clauses with atoms less than or equal to the bound are unsatisfiable (i.e., $\exists \beta.\ \nexists \mathcal{I}.\ \mathcal{I} \models_{\mathsf{clss}} (\mathsf{gnd}^{\preceq_B \beta}_{\mathsf{clss}}\ N)$).*

Note that while Lemma 5.23 proves the existence of an appropriate bound, it provides no constructive way of finding one. What one can do is follow along Theorem 5.6 and iteratively increase a heuristically chosen bound until an appropriate one is found; if the set of initial clauses is unsatisfiable, this will terminate.

**Remark 5.24.** *Lemma 5.23's hypothesis that $\prec_B$ is a well-founded, total, strict ordering cannot be expressed as a theorem-local hypothesis. The reason is that the compactness theorem for clausal first-order logic requires terms to be an instance of the* wellorder *type class, which is not the case in the* scl_fol_calculus *locale, where the assumptions on the $\prec_B$ relation are kept minimal. Because Isabelle does not allow to instantiate a type class with a concrete type inside a locale or theorem, we define a new locale that extends* scl_fol_calculus *and adds a type class requirement on the first-order term constants. This enables the type-class system to automatically instantiate the* wellorder *type class for terms using the previously registered Knuth–Bendix ordering. We then instantiate the $\prec_B$ relation of* scl_fol_calculus *with the Knuth–Bendix ordering. This type class and locale gymnastic could be avoided if the formalization of the compactness theorem was refactored to offer a predicate-based version alongside the existing type-class-based version.*

## 5.4 Conclusion

We generalized and formalized the SCL(FOL) calculus in Isabelle/HOL. The main results are formal proofs of soundness, nonredundancy of learned clauses, termination, and refutational completeness. Because the formalization was performed simultaneously to Bromberger et al.'s pen-and-paper work [29], they could benefit from each other. A mechanized formalization must consider low-level details, but it is also the opportunity to identify the most important aspects of the theory and abstract over details needed in the context of an actual implementation. For example, we abstracted from the level of a state to define the Backtracking rule and replaced it with an abstract specification of the result. In contrast, a level was used in all pen-and-paper presentations of the calculus to have a constructive way of going back to the maximal trail where the learned clause propagates. The abstraction supports investigation of several Backtrack rule versions and to base the soundness result on a version with a minimal requirement (i.e., the learned clause is no longer false with respect to the trail).

The formalization did uncover a small bug in the calculus, but also showed that its effect was very localized and naturally lead to a solution. Another benefit of the

formalization is how much it supports refactoring and exploratory experimentation. When making a change to a definition or a conjecture, Isabelle immediately and exhaustively points to the parts that need to be adapted. Very often, proofs can automatically be adapted using proof automation tools such as Sledgehammer. This was invaluable to quickly try out ideas or change subtle parts of the calculus. One such example is in the Resolve rule, where the formalization first used substitution composition as found in the original calculus and latter replaced it by an abstract specification of merged grounding. This idea came from a private discussion sketching an eventual C implementation where it became clear that substitution composition would be a costly operation. We then introduced the abstract specification of merged grounding and fixed the formalization by following the mistakes reported by Isabelle.

# Part II

# Simulations between Calculi

# Chapter 6

# A Framework for Simulation Proofs

This chapter is based on two workshop papers coauthored with Stefan Brunthaler [44, 45]. The two papers describe my formalization work.

## 6.1 Introduction

The mechanically verified formalization of software components has been the subject of much research in the last decades. Especially influential were the CompCert [74] and CakeML [71] projects, which produced realistic compilers from (a large subset of) two real-world programming languages (C99 and Standard ML) to real hardware platforms. These compilers showed both that mechanized verification is feasible and that it has a measurable effect on the dependability of the compiler [127].

We can now observe a shift in perspective, where the idea of mechanically verified software components is becoming a concrete and desirable goal. Formalization projects are increasing in number, but also in size, complexity, and lifetime. There is an analogy to be made with the emergence, in the second half of the 20th century, of software engineering to the point that the term *proof engineering* starts to be used. New and interesting questions now emerge. How to avoid repetition in definitions and proofs? Which concepts can be generalized and reused? How to separate a formalization in independent components, so that multiple people can work in parallel? What should be the interface between such components? How can tooling make proof engineers more productive? What is a good balance between proof readability and the time required to (mechanically) verify it? etc.

In the case of compiler verification, we have a very well-understood domain, with well-known terminology, that builds on decades of research and empirical experience. But as is the case for a lot of small software prototypes, small-scale formalizations constantly redefined similar abstractions and concepts. This is something we wanted to avoid when we started the formalization in Isabelle/HOL [85] of three small stack-based interpreters for bytecode languages implementing different run-time optimizations. Inspired by the concept of modularization in software engineering, we separated the general concepts from the language-dependent parts. We learned about, and made use of, Isabelle's locales to devise a small generic framework for the

verification of program transformations. Our framework is available in the *Archive of Formal Proofs* [38].

## 6.2 Background

The operational semantics of a programming language can be defined as a transition system representing the execution of a program written in this language. A language $L = \langle S, I, F, \rightarrow \rangle$ is defined by a set $S$ of program states, a set $I \subseteq S$ of initial states, a set $F \subseteq S$ of final states, and a transition relation $\rightarrow \subseteq S \times S$. The execution of a program is modelled as a sequence of states $s_1 \rightarrow s_2 \rightarrow ...$ with $s_1 \in I$. An execution is called terminating if there exists a state $s_n$ such that $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ and $\nexists s_{n+1}.\ s_n \rightarrow s_{n+1}$, and nonterminating otherwise. A terminating execution is said to be successful if $s_n \in F$ and to go wrong otherwise. These execution behaviours are usually called the program's behaviour and written $s \Downarrow b$.

The compiler from a language $L_1$ to $L_2$ is a partial function $\mathcal{C} : S_1 \rightharpoonup S_2$, which maps a program $s \in I_1$ to $\mathcal{C}(s) \in I_2$.

Two programs $s$ and $c$ are said to be equivalent if they exhibit the same behaviour, i.e., $\forall b.\ s \Downarrow b \longleftrightarrow c \Downarrow b$. This can be established using a bisimulation [96]: the conjunction of a backward and a forward simulation. Consider a binary relation $\approx$, between program states, expressing that two states are to be considered equivalent for a given use case. This relation is called a lockstep simulation whenever

$$\forall s\ s'\ c.\ s \approx c \wedge s \rightarrow s' \longrightarrow (\exists c'.\ s' \approx c' \wedge c \rightarrow c')$$

holds. A backward simulation, thus, shows that every behaviour of the compiled program is also a behaviour of the source program, i.e., the compilation is correct (sound). A forward simulation shows that every behaviour of the source program can be achieved by the compiled program (i.e., the compilation is complete).

## 6.3 The Design of the Framework

The framework has three main components: some abstract definitions of languages and compilers using locales, a generic definition of program behaviour, and some composition operations over simulations and compilers. Locales are Isabelle's module system to define hierarchies of parametric theories [8]; they are described in Section 2.5.

### 6.3.1 Locales

The definition of programming languages is separated into two parts: an abstract semantics and a concrete program representation.

```
locale semantics =
  fixes step :: 's ⇒ 's ⇒ bool and final :: 's ⇒ bool
  assumes ∀s. final s ⟶ (∄s'. step s s')

locale language = semantics step final
  for step :: 's ⇒ 's ⇒ bool and final :: 's ⇒ bool +
  fixes load :: 'p ⇒ 's ⇒ bool
```

The semantics locale represents the semantics as an abstract machine. It is expressed by a transition system on states (type variable $'s$) with a transition relation *step*—often written as an infix short arrow—and a predicate *final* identifying final states. The semantics locale assumes that final states cannot have any more transition.

The language locale represents the concrete program representation (type variable $'p$), which can be associated to a program state by the relation *load*; it is meant to represent initialization (e.g., of a list of arguments given at the command line to the program) and nondeterminism (e.g., by initializing a pseudo-random number generator) when loading a program. The set of initial states of the transition system is implicitly defined by the set of program states on which the relation *load* is defined (i.e., $\{s \mid \exists p.\ load\ p\ s\}$).

The similarity of two given semantics is expressed formally through simulations.

**locale** forward_simulation =
  $\mathsf{L}_1$: semantics $step_1$ $final_1$ + $\mathsf{L}_2$: semantics $step_2$ $final_2$
  **for**
    $step_1 :: 's_1 \Rightarrow 's_1 \Rightarrow bool$ **and** $final_1 :: 's_1 \Rightarrow bool$ **and**
    $step_2 :: 's_2 \Rightarrow 's_2 \Rightarrow bool$ **and** $final_2 :: 's_2 \Rightarrow bool$
  **fixes**
    $match :: 'i \Rightarrow 's_1 \Rightarrow 's_2 \Rightarrow bool$ **and**
    $(\prec) :: 'i \Rightarrow 'i \Rightarrow bool$
  **assumes**
    wfp $(\prec)$ **and**
    $\forall i\ s_1\ s_2.\ match\ i\ s_1\ s_2 \longrightarrow final_1\ s_1 \longrightarrow final_2\ s_2$ **and**
    $\forall i\ s_1\ s_1'\ s_2.\ match\ i\ s_1\ s_2 \longrightarrow step_1\ s_1\ s_1' \longrightarrow$
      $(\exists i'\ s_2'.\ {step_2}^+\ s_2\ s_2' \wedge match\ i'\ s_1'\ s_2') \vee (\exists i'.\ match\ i'\ s_1'\ s_2 \wedge i' \prec i)$

A forward simulation is defined between two semantics $\mathsf{L}_1$ and $\mathsf{L}_2$. It is parametrized by a predicate *match*, which expresses that two states from $\mathsf{L}_1$ (type $'s_1$) and $\mathsf{L}_2$ (type $'s_2$) are considered equivalent w.r.t. some annotation (type $'i$), and a relation $\sqsubset$ on annotations. The annotation on the predicate *match* is a measurement used to avoid stuttering (i.e., one transition system progresses while the other does not).

The assumptions of a forward simulation are that $\prec$ is a well-founded relation, that a final state of $\mathsf{L}_1$ corresponds to a final state of $\mathsf{L}_2$, and that a step of $\mathsf{L}_1$ either represents a nonempty sequence of steps of $\mathsf{L}_2$—the ${step_2}^+$ relation is the transitive closure of $step_2$—or results in an equivalent state. Stuttering is ruled out by the requirement that the measurement of the *match* predicate decreases w.r.t. $\prec$.

**locale** backward_simulation =
  $\mathsf{L}_1$: semantics $step_1$ $final_1$ + $\mathsf{L}_2$: semantics $step_2$ $final_2$
  **for**
    $step_1 :: 's_1 \Rightarrow 's_1 \Rightarrow bool$ **and** $final_1 :: 's_1 \Rightarrow bool$ **and**
    $step_2 :: 's_2 \Rightarrow 's_2 \Rightarrow bool$ **and** $final_2 :: 's_2 \Rightarrow bool$
  **fixes**
    $match :: 'i \Rightarrow 's_1 \Rightarrow 's_2 \Rightarrow bool$ **and**
    $(\sqsubset) :: 'i \Rightarrow 'i \Rightarrow bool$
  **assumes**
    wfp $(\sqsubset)$ **and**

$\forall i\ s_1\ s_2.\ match\ i\ s_1\ s_2 \longrightarrow final_2\ s_2 \longrightarrow final_1\ s_1$ **and**
$\forall i\ s_1\ s_2\ s_2'.\ match\ i\ s_1\ s_2 \longrightarrow step_2\ s_2\ s_2' \longrightarrow$
$(\exists i'\ s_1'.\ step_1{}^+\ s_1\ s_1' \wedge match\ i'\ s_1'\ s_2') \vee (\exists i'.\ match\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

A backward simulation is defined similarly to a forward simulation except that the occurrences of $step_1$ and $step_2$ as well as $final_1$ and $final_2$ are swapped in the assumptions. For the rest of this chapter, we will focus on results shown from the perspective of a backward simulation; many results hold dually for forward simulations.

**locale** bisimulation =
forward_simulation $step_1\ final_1\ step_2\ final_2\ match\ (\prec)$ +
backward_simulation $step_1\ final_1\ step_2\ final_2\ match\ (\sqsubset)$ +
**for**
$step_1 :: 's_1 \Rightarrow 's_1 \Rightarrow bool$ **and** $final_1 :: 's_1 \Rightarrow bool$ **and**
$step_2 :: 's_2 \Rightarrow 's_2 \Rightarrow bool$ **and** $final_2 :: 's_2 \Rightarrow bool$ **and**
$match :: 'i \Rightarrow 's_1 \Rightarrow 's_2 \Rightarrow bool$ **and**
$(\sqsubset) :: 'i \Rightarrow 'i \Rightarrow bool$ **and**
$(\prec) :: 'i \Rightarrow 'i \Rightarrow bool$

A bisimulation is composed of both a forward and backward simulation w.r.t. the same matching relation. Note that the well-founded relation $\prec$, used in the forward relation, and the well-founded relation $\sqsubset$, used in the backward simulation, must work on the same annotations of the matching relation.

**locale** compiler =
$\mathsf{L}_1$: language $step_1\ final_1\ load_1$ + $\mathsf{L}_2$: language $step_2\ final_2\ load_2$ +
backward_simulation $step_1\ final_1\ step_2\ final_2\ match\ (\sqsubset)$
**for**
$step_1 :: 's_1 \Rightarrow 's_1 \Rightarrow bool$ **and** $final_1 :: 's_1 \Rightarrow bool$ **and**
$load_1 :: 'p_1 \Rightarrow 's_1 \Rightarrow bool$ **and**
$step_2 :: 's_2 \Rightarrow 's_2 \Rightarrow bool$ **and** $final_2 :: 's_2 \Rightarrow bool$ **and**
$load_2 :: 'p_2 \Rightarrow 's_2 \Rightarrow bool$ **and**
$match :: 'i \Rightarrow 's_1 \Rightarrow 's_2 \Rightarrow bool$ **and**
$(\sqsubset) :: 'i \Rightarrow 'i \Rightarrow bool$ +
**fixes** $compile :: 'p_1 \Rightarrow 'p_2\ option$
**assumes**
$\forall p_1\ p_2\ s_2.\ compile\ p_1 = p_2 \longrightarrow load_2\ p_2\ s_2 \longrightarrow$
$(\exists s_1\ i.\ load_1\ p_1\ s_1 \wedge match\ i\ s_1\ s_2)$

The compiler locale relates two languages, $\mathsf{L}_1$ and $\mathsf{L}_2$, by a backward simulation and provides a (deterministic) partial function *compile* that maps a concrete program of $\mathsf{L}_1$ to a concrete program of $\mathsf{L}_2$. The only assumption is that a successful compilation results in a program which, when loaded, is equivalent to the loaded initial program.

### 6.3.2 Behaviours

We define a generic datatype to encode three broad execution behaviours: successful termination (constructor Terminates), nonterminating execution (constructor

Diverges), and erroneous termination (constructor Goes_wrong).

**datatype** $'s$ *behaviour* $=$ Terminates $'s$ | Diverges | Goes_wrong $'s$

Successfully and erroneously terminating behaviours are annotated with the last state of the execution to compare the result of two executions with the rel_behaviour :: $('s \Rightarrow 's \Rightarrow bool) \Rightarrow 's$ *behaviour* $\Rightarrow 's$ *behaviour* $\Rightarrow$ *bool* relation.

$\forall R\ s_1\ s_2.\ R\ s_1\ s_2 \longrightarrow$ rel_behaviour $R$ (Terminate $s_1$) (Terminate $s_2$)
$\forall R\ s_1\ s_2.$ rel_behaviour $R$ Diverges Diverges
$\forall R\ s_1\ s_2.\ R\ s_1\ s_2 \longrightarrow$ rel_behaviour $R$ (Goes_wrong $s_1$) (Goes_wrong $s_2$)

The exact meaning of the three behaviours is defined in the semantics locale, where a binary relation $(\downarrow) :: 's \Rightarrow 's$ *behaviour* $\Rightarrow$ *bool* is defined to assign an execution behaviour to a program state. Note that $step^*$ is the reflexive transitive closure of *step* and that $step^\infty$ is its infinitely transitive closure.

$\forall s\ s'.\ step^*\ s\ s' \longrightarrow (\nexists s''.\ step\ s'\ s'') \longrightarrow final\ s' \longrightarrow s \downarrow$ (Termination $s'$)
$\forall s.\ step^\infty\ s \longrightarrow s \downarrow$ Diverges
$\forall s\ s'.\ step^*\ s\ s' \longrightarrow (\nexists s''.\ step\ s'\ s'') \longrightarrow \neg final\ s' \longrightarrow s \downarrow$ (Goes_wrong $s'$)

We lift the relation $\downarrow$ on program states to a relation $(\Downarrow) :: 'p \Rightarrow 's$*behaviour* $\Rightarrow$ *bool* on concrete program representations.

$$\forall p\ b.\ p \Downarrow b \longleftrightarrow (\exists s.\ load\ p\ s \wedge s \downarrow b)$$

In general, the transition relation *step* in the semantics locale does not have to be deterministic; but if it is, then the behaviour of a program state is also deterministic.

**Lemma 6.1.** *If step is right-unique (i.e.,* $\forall s\ s'\ s''.\ step\ s\ s' \longrightarrow step\ s\ s'' \longrightarrow s' = s''$*), then* $\downarrow$ *is right-unique (i.e.,* $\forall p\ b\ b'.\ p \downarrow b \longrightarrow p \downarrow b' \longrightarrow b = b'$*).*

If the relation *load* in the language locale is also deterministic, then the behaviour of a concrete program representation is also deterministic.

**Lemma 6.2.** *If step and load are both right-unique, then* $\Downarrow$ *is right-unique.*

The main correctness theorem for backward simulations states that, for any two matching programs, any not wrong behaviour of the later is also a behaviour of the former. In other words, if the compiled program does not crash, then its behaviour, whether it terminates or not, is a also a valid behaviour of the source program.

**Theorem 6.3.** *Let* $s_1$ *and* $s_2$ *be program states of* $\mathsf{L}_1$ *and* $\mathsf{L}_2$ *respectively. If* $s_1$ *matches* $s_2$ *(i.e.,* $\exists i.$ *match* $i\ s_1\ s_2$*),* $s_2$ *has behaviour* $b_2$ *(i.e.,* $s_2 \downarrow b_2$*), and* $b_2$ *is not a wrong behaviour, then* $s_1$ *has a behaviour that matches with* $b_2$ *(i.e.,* $\exists b_1\ i'.\ s_1 \downarrow b_1 \wedge$ rel_behaviour $(match\ i')\ b_1\ b_2$*).*

Because this theorem is proven in the context of the backward_simulation locale and, thus, only depends on its parameters and assumptions, it is independent of the concrete programming language and needs to be proven only once. It automatically holds for all interpretations of the locale.

As a corollary, the preservation of behaviour can be lifted to the compilation of a concrete program representation.

**Corollary 6.4.** *Let $p_1$ and $p_2$ be concrete program representations of* $\mathsf{L}_1$ *and* $\mathsf{L}_2$ *respectively. If $p_1$ compiles to $p_2$ (i.e., compile $p_1 = p_2$), $p_2$ has behaviour $b_2$ (i.e., $s_2 \Downarrow b_2$), and $b_2$ is not a wrong behaviour, then $p_1$ has a behaviour that matches with $b_2$ (i.e., $\exists b_1\ i'.\ p_1 \Downarrow b_1 \wedge$* rel_behaviour *(match $i'$) $b_1$ $b_2$).*

### 6.3.3 Generic Composition of Simulations and Compilers

We define the generic composition of matching functions, $(\diamond) :: ('i \Rightarrow 'a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('j \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'i \times 'j \Rightarrow 'a \Rightarrow 'c \Rightarrow bool$, and orderings, $\mathsf{lex}_{\mathsf{prod}} :: ('i \Rightarrow 'i \Rightarrow bool) \Rightarrow ('j \Rightarrow 'j \Rightarrow bool) \Rightarrow 'i \times 'j \Rightarrow 'i \times 'j \Rightarrow bool$, such that the composition of two backward simulations is itself a backward simulation.

**Lemma 6.5.** *Let $\mathcal{S}_1 = \langle step_1; final_1 \rangle$, $\mathcal{S}_2 = \langle step_2; final_2 \rangle$, and $\mathcal{S}_3 = \langle step_3; final_3 \rangle$ be semantics. Let $\sim$ and $\approx$ be matching relations. Let $\prec$ and $\sqsubset$ be relations used to limit stuttering. If*

- *there is a backward simulation between $\mathcal{S}_1$ and $\mathcal{S}_2$ w.r.t. $\sim$ and $\prec$:*
  backward_simulation *$step_1$ $final_1$ $step_2$ $final_2$ $(\sim)$ $(\prec)$*
- *there is a backward simulation between $\mathcal{S}_2$ and $\mathcal{S}_3$ w.r.t. $\approx$ and $\sqsubset$:*
  backward_simulation *$step_2$ $final_2$ $step_3$ $final_3$ $(\approx)$ $(\sqsubset)$*

*then there is a backward simulation between $\mathcal{S}_1$ and $\mathcal{S}_3$ w.r.t. $(\sim) \diamond (\approx)$ and* $\mathsf{lex}_{\mathsf{prod}}$ *$(\prec)$ $(\sqsubset)$:*
backward_simulation *$step_1$ $final_1$ $step_3$ $final_3$ $((\sim) \diamond (\approx))$* ($\mathsf{lex}_{\mathsf{prod}}$ *$(\prec)$ $(\sqsubset)$).*

We define the generic $\circledcirc :: ('b \Rightarrow 'c\ option) \Rightarrow ('a \Rightarrow 'b\ option) \Rightarrow 'a \Rightarrow 'c\ option$ composition operator on compilers, which corresponds to the monadic bind of the *option* type found in a compiler's codomain.

$$\forall C_1\ C_2\ p.\ (C_2 \circledcirc C_1)\ p = \mathsf{Option.bind}\ (C_1\ p)\ C_2$$

Its correctness can then be generically proven for any two interpretations of the compiler locale.

**Theorem 6.6.** *Let $\mathcal{L}_1 = \langle step_1; final_1; load_1 \rangle$, $\mathcal{L}_2 = \langle step_2; final_2; load_2 \rangle$, and $\mathcal{L}_3 = \langle step_3; final_3; load_3 \rangle$ be languages. Let $C_1$ and $C_2$ be compilation functions. Let $\sim$ and $\approx$ be matching relations. Let $\prec$ and $\sqsubset$ be relations used to limit stuttering. If*

- *$C_1$ is a compiler from $\mathcal{L}_1$ to $\mathcal{L}_2$ w.r.t. $\sim$ and $\prec$:*
  compiler *$step_1$ $final_1$ $load_1$ $step_2$ $final_2$ $load_2$ $(\sim)$ $(\prec)$ $C_1$*
- *$C_2$ is a compiler from $\mathcal{L}_2$ to $\mathcal{L}_3$ w.r.t. $\approx$ and $\sqsubset$:*
  compiler *$step_2$ $final_2$ $load_2$ $step_3$ $final_3$ $load_3$ $(\approx)$ $(\sqsubset)$ $C_2$*

*then $C_2 \circledcirc C_1$ is a compiler from $\mathcal{L}_1$ to $\mathcal{L}_3$ w.r.t. $(\sim) \diamond (\approx)$ and* $\mathsf{lex}_{\mathsf{prod}}$ *$(\prec)$ $(\sqsubset)$:*
compiler *$step_1$ $final_1$ $load_1$ $step_3$ $final_3$ $load_3$ $((\sim) \diamond (\approx))$* ($\mathsf{lex}_{\mathsf{prod}}$ *$(\prec)$ $(\sqsubset)$)$(C_2 \circledcirc C_1)$.*

## 6.4 Discussion

Using locales as a modularization tool for our generic framework turned out to be elegant at times and frustrating in other cases.

### 6.4.1 Strengths of the Approach

**Parameters, assumptions, and derived elements are clearly separated.** The syntax used to define a locale enables the user to clearly state the parameters and assumptions that are abstracted over. Derived elements such as function definitions and lemmas are clearly separated by being defined later in a locale context. The fact that these extensions can be done at any point following the locale's definition gives a lot of flexibility when structuring the formalization.

**It is possible to abstract over multiple types.** Locales enable parameters to depend on multiple type variables. This makes them more general than type classes, with which they have otherwise a lot in common. While traditional type classes permit to abstract over operations on a given abstract type, locales permit to abstract over both operations on concrete types and multiple abstract types. In fact, type classes in Isabelle/HOL are just syntactic sugar for locales with a single type variable.

**It is possible to have multiple interpretations for a given set of type.** Because a locale interpretation introduces a new namespace when specializing the derived elements, multiple instantiations are possible for a given set of types. A classical example for such a situation is a partial ordering over the integers. Using traditional type classes, one has to decide a canonical ordering that will be associated with the integer type. To use an alternate ordering, one has to define a bijection to an alternative type which instantiate the type class accordingly. As many distinct types and bijections are required as distinct instantiations are wished.

### 6.4.2 Weaknesses of the Approach

**Parametric types and type aliases cannot be defined in locales.** This limitation requires the user to generalize data types to abstract over any type variable fixed in the locale definition and define them outside of the locale. This generalization is trivial, because a fixed type variable in a locale is akin to a type variable in a data type definition. The burden shows up when referring to the generic type in a type annotation, where it must be explicitly instantiated. Because parametric type aliases are also not supported, the instantiation has to be repeated over and over. As the number of type variables increases, type annotations become more complex, longer to write, and harder to read.

**When extending existing locales, type annotations on fixed variables are required to name type variables.** These variables appear in the **for** section and their types are inferred from their usage in instantiating the extended locales to be extended. Type inference even succeeds in cases some type variables must be unified between multiple locale instantiations, as is the case in the *compiler* locale. The user must nevertheless provide some type annotations in order to name the type variables that will be referred later. In practice, most of them are requires in type annotations.

**Proving lemmas involving locale predicates has considerable syntactic overhead.** Consider Theorem 6.6, where the two hypotheses and the conclusion are locale predicates of the compiler locale. The proof involves accessing the language instance predicates accessible with expressions, such as *assms*(1)[THEN compiler.axioms(1)]. The problem with this syntax is twofold: (i) it depends on the order in which the axioms were stated, and (ii) it does not scale well when the user needs to extract multiple axioms from multiple assumptions. The first problem could be solved by automatically adding lemmas using the name of the extension (e.g., compiler.$\mathsf{L}_1$ could be a synonym of compiler.axioms(1) to refer to the first language instance predicate of the compiler's definition). The second could be alleviated if unnamed contexts supported locales extension.

**References to a locale's fixed variables and derived definitions are syntactically different.** When extending locales, as is the case in the backward_simulation locale, derived definitions of the two languages are accessible with uniform names in some namespaces, such as $\mathsf{L}_1$.behaves and $\mathsf{L}_2$.behaves. Fixed parameters, by contrast, are only accessible using their given name, e.g., *step1* and *step2*. Even though explicitly naming locale parameters may be omitted in simple cases, it is required as soon as two locales fix parameters with the same name. While writing locales for software abstractions, as opposed to mathematical structures, we observed that fixed parameters must be named in all but the most trivial cases.

The lack of a uniform syntax to access derived definitions and parameters also has an undesired impact on refactoring. If one of the locale parameters gets replaced by an equivalent derived definition, not only do all interpretations of the locale have to be adapted, but all definitions and theorems derived in a locale identified by a prefixed name must also be adapted (e.g., everything depending on $\mathsf{L}_1$ or $\mathsf{L}_2$ in the backward_simulation locale). Although this would not necessarily be a problem in the absence of name clashes, a uniform naming scheme allowing the systematical use of the interpretation's prefix would benefit both the locale's design and implementation.

**The syntax overhead of locale extension increases with the number of fixed parameters or types.** This can already be observed in the definition of the compiler locale where the **for** section specifies the signature of eight parameters coming from the extended locales; only one new parameter in introduced in the **fixes** section. The current syntax seems to serve two purposes: (i) provide unique names for fixed parameters, and (ii) state which abstract types are shared in parameters' types. To satisfy the first purpose, one could name the locale instances and use the uniform naming scheme mentioned above, thereby increasing their utility and applicability. To satisfy the second purpose, the locale mechanism should offer an alternative syntax that allows for a more succinct way to express type dependencies between different locale extensions.

## 6.5 Conclusion

We presented a generic framework for formalizing compilers in Isabelle/HOL. It is based on locales to abstract over the concrete languages and program transformations, provides general definitions for program behaviours and compiler compositions, and

generically proves preservation of behaviour. This generality makes it usable for in other use cases manipulating transition systems; we present such use case in Chapter 7 where we prove the equivalence of two logical calculi.

This framework emerged as a side product of our formalization of three stack-based languages that implement different optimizations; this formalization is presented in Chapter 8. It helped us to emphasis the commonalities between the different theories and to reduce some duplication. Possible future work includes extending the semantics to support traces, exploring how to extract executable programs from such formalizations, and exploring how to simplify or automate repetitive operations such as the composition of multiple compilers.

We also reported on our experience using locales as an abstraction mechanism and highlighted how locales could be improved to better serve such software formalization. On the one hand, the additional modularity and abstraction afforded by locales to structure proof developments is an enormous benefit. On the other hand, the syntactic overhead makes them cumbersome to use for concepts built from lots of parameters; it also makes refactoring a tedious tasks, as every usage of the locale's content could potentially need to be adapted.

# Chapter 7

# Simulation between SCL(FOL) and Ground Ordered Resolution

This chapter describes unpublished work coproduced with Martin Bromberger and Christoph Weidenbach. The refinement proof was designed on paper by me with help from Martin Bromberger. The Isabelle/HOL formalization was made by me in parallel to the proof design.

## 7.1 Introduction

The SCL ("Clause Learning from Simple Models" or simply "Simple Clause Learning") family of calculi lifts a conflict-driven clause learning (CDCL) approach [12, 82] to first-order logic [26, 28, 29, 52, 72]. SCL(FOL), a member of this family, is a calculus for first-order logic without equality that works by refutation and model construction. It operates on a clause set, which initially consists of the clausified input problem in which the conjecture appears negated. It iteratively and heuristically builds a ground (i.e., variable-free) candidate model for the clause set. Being heuristically built, the candidate model might at some point conflict with one of the clauses. Then, SCL(FOL) infers a new nonground clause from the candidate model and the conflicting clause, adds this new clause to the set of known clauses, and backtracks to a smaller, conflict-free candidate model. The calculus stops either when the empty clause $\bot$, denoting falsehood, is derived or when the candidate model makes the ground instances of the clauses true that are less than a fixed bound. SCL(FOL) was proven to be both sound and refutationally complete, meaning that it will eventually derive $\bot$ for any unsatisfiable clause set, and its learned clauses were proven to always be nonredundant. The proofs were carried out first on paper [52] and later in the Isabelle/HOL proof assistant [25].

Being a relatively new calculus, SCL(FOL) is often compared with *resolution*, which is likely the most well-known refutationally complete calculus for first-order logic without equality. Resolution works by refutation and saturation of a clause set of the clausified input problem. Inferences are performed using clauses from this set as premises; the conclusions of inferences are added to the set. The calculus stops either when the empty clause $\bot$ is derived or when no more inferences are possible. *Ordered resolution* is a refinement of resolution where a term ordering is used to significantly reduce the number of possible inferences while still preserving

refutational completeness; it is sometimes also called superposition without equality. *Ground ordered resolution* is a version of the calculus restricted to work only with ground (i.e., variable-free) clauses.

Bromberger, Jain, and Weidenbach recently showed that SCL(FOL) can simulate the derivation of nonredundant clauses by ground ordered resolution [27]. This not only brought new insights on the relationship between the calculi, but also showed that SCL(FOL)'s flexible model construction can be tuned to build the exact same model as the one built in ordered resolution's proof of refutational completeness. Proving this simulation was an ambitious endeavor because the two calculi work quite differently. The main difference is that ordered resolution can apply its inference rules relatively freely, learning many redundant clauses along the way, while SCL(FOL)'s inferences are strictly guided by conflict resolution and can only lead to learning nonredundant clauses. To bridge this gap, Bromberger et al. first specified a deterministic strategy SUP-MO for ordered resolution that only learns nonredundant clauses. The strategy uses the model construction operation found in ordered resolution's proof of refutational completeness because it is relatively simple and well-known in the automated reasoning community. They then specified a corresponding deterministic strategy SCL-SUP for SCL(FOL) and showed that SCL-SUP simulates SUP-MO.

Consider the example from Bromberger et al. of proving that the formulas $\mathsf{P}(\mathsf{a}) \vee \mathsf{P}(\mathsf{a})$ and $\mathsf{P}(\mathsf{a}) \longrightarrow \mathsf{Q}(\mathsf{b})$ entail the formula $\mathsf{Q}(\mathsf{b})$. After clausification and negation of the conjecture, we obtain the clause set $N = \{\mathsf{P}(\mathsf{a}) \vee \mathsf{P}(\mathsf{a}), \neg\mathsf{P}(\mathsf{a}) \vee \mathsf{Q}(\mathsf{b}), \neg\mathsf{Q}(\mathsf{b})\}$. We assume the symbol precedence $\mathsf{a} \prec \mathsf{b} \prec \mathsf{P} \prec \mathsf{Q}$ that we lift to terms, atoms, literals, and clauses.

The strategy SUP-MO for ordered resolution first constructs the candidate model $\{\}$, meaning that no atom is considered true, and finds the least clause false w.r.t. the model: $\mathsf{P}(\mathsf{a}) \vee \mathsf{P}(\mathsf{a})$; it then uses the factorization rule to infer the clause $\mathsf{P}(\mathsf{a})$. At this point, the clause set becomes $N \cup \{\mathsf{P}(\mathsf{a})\}$. SUP-MO then constructs a new candidate model $\{\mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{b})\}$, meaning that only the atoms $\mathsf{P}(\mathsf{a})$ and $\mathsf{Q}(\mathsf{b})$ are considered true, and finds the least clause false w.r.t. the model: $\neg\mathsf{Q}(\mathsf{b})$; it then uses the resolution rule on $\neg\mathsf{P}(\mathsf{a}) \vee \mathsf{Q}(\mathsf{b})$ and $\neg\mathsf{Q}(\mathsf{b})$ to infer the clause $\neg\mathsf{P}(\mathsf{a})$. At this point, the clause set becomes $N \cup \{\mathsf{P}(\mathsf{a}), \neg\mathsf{P}(\mathsf{a})\}$. SUP-MO finally constructs the new candidate model $\{\mathsf{P}(\mathsf{a})\}$ and finds the least clause false w.r.t. the model: $\neg\mathsf{P}(\mathsf{a})$; it then uses the resolution rule on $\mathsf{P}(\mathsf{a})$ and $\neg\mathsf{P}(\mathsf{a})$ to infer the empty clause $\bot$. This concludes the run of the strategy SUP-MO. It was able to derive $\bot$ from the clausified problem $N$, which means that the formulas $\mathsf{P}(\mathsf{a}) \vee \mathsf{P}(\mathsf{a})$ and $\mathsf{P}(\mathsf{a}) \longrightarrow \mathsf{Q}(\mathsf{b})$ do entail the formula $\mathsf{Q}(\mathsf{b})$.

The strategy SCL-SUP for SCL(FOL) aims to perform the same inferences in the same order as SUP-MO. Because SCL(FOL) can only factorize clauses in the context of resolution inferences, it does not learn the clauses that SUP-MO inferred through factorization, but it still keeps track of them to act as if they had been learned. SCL-SUP first iteratively builds the candidate model $[\mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{b})]$, keeping track that the clause $\mathsf{P}(\mathsf{a})$ can be inferred by $\mathsf{P}(\mathsf{a}) \vee \mathsf{P}(\mathsf{a})$, and then detects a conflict with the clause $\neg\mathsf{Q}(\mathsf{b})$; it performs conflict resolution, learns the clause $\neg\mathsf{P}(\mathsf{a})$, and backtracks to the empty candidate model $\epsilon$. At this point, the clause set becomes $N \cup \{\neg\mathsf{P}(\mathsf{a})\}$. SCL-SUP then builds the candidate model $[\mathsf{P}(\mathsf{a})]$ and then detects a conflict with the clause $\neg\mathsf{P}(\mathsf{a})$; it performs conflict resolution, infers the empty clause

$\bot$, and backtracks to the empty candidate model $\epsilon$.

In summary, SCL-SUP closely follows SUP-MO. It takes note of all clauses inferred through factorization by SUP-MO and learns all clauses inferred through resolution by SUP-MO. This note taking and clause learning is performed in the exact same order as the inferences in SUP-MO. The candidate model explicitly built by SCL-SUP also corresponds to the model construction implicitly performed by SUP-MO.

Bromberger et al.'s contribution was significant but, due to the space constraints of a 16-page paper, the published proof is monolithic and hard to comprehend. They also proved a simulation in only one direction even though it should be possible to also prove the other direction.

In this chapter, we revisit the simulation between the ground ordered resolution and SCL(FOL) calculi. We reuse the existing strategy for ground ordered resolution but provide a new strategy for SCL(FOL) and a new simulation proof. Our contributions are as follows.

1. We specified a new, simpler strategy for SCL(FOL) that simulates ground ordered resolution learning nonredundant clauses. The strategy is expressed as a simple transition relation closely following SCL(FOL)'s transition relation.
2. We provided a stronger bisimulation proof between SUP-MO and our new strategy. Our proof is modular: it consists of ten refinement steps focusing on different aspects of the two strategies.
3. We provided a reusable lifting lemma that can lift a simulation between any well-behaved transition systems to a bisimulation.
4. We formalized all the above in Isabelle/HOL.

Our Isabelle formalization consists of approximately 26 000 nonblank lines[1] and is available in the *Archive of Formal Proofs* [43]. Our work is part of the IsaFoL (Isabelle Formalization of Logic) effort [18].

## 7.2 Preliminaries

Before proving simulations between SCL(FOL) and ground ordered resolution, we first need a formalization of the semantics of SCL(FOL), which we presented in Chapter 5, a formalization of the semantics of ground ordered resolution (Section 7.2.1), and a formal definition of simulation (Section 7.2.2).

### 7.2.1 Ground Ordered Resolution

For this project, we need a formalization of ground ordered resolution and it turns out that the superposition calculus presented in Chapter 4 is an extension of ordered resolution and that Section 4.4 formalizes ground superposition.

We used the crude but very effective method of manually extracting a formalization of ground ordered resolution out of our existing formalization of ground superposition. We basically copied the Isabelle theory file, removed everything related to equality,

[1] Counted using `grep -Ev '^[[:blank:]]*$'`.

repaired the broken proofs, and renamed all occurrences of the word superposition by ordered resolution.

This new formalization defines the inference rules, the standard redundancy criterion, and the canonical model construction operation. It also proves soundness and refutational completeness of the calculus.

For the sake of brevity, we will only present the definitions of the inference rules and model construction, and refer the interested reader to the Isabelle formalization for more details.

Ground ordered resolution only has two inference rules: resolution and factoring. The ground resolution rule is as follows. The rule notation below defines an inductive predicate $\mathsf{ground\_resolution}\ D\ E\ C$ with the rule's premises $D$ and $E$ as assumptions and the inferred clause $C$ as conclusion:

$$\frac{\overbrace{A \vee D'}^{D} \qquad \overbrace{\neg A \vee E'}^{E}}{\underbrace{D' \vee E'}_{C}}\ \mathsf{ground\_resolution}\ D\ E\ C$$

Side conditions:

1. $D \prec_{\mathsf{cls}} E$
2. either $sel_{\mathsf{G}}\ E = \{\}$ and $\neg A$ is maximal in $E$ or $\neg A \in sel_{\mathsf{G}}\ E$
3. $sel_{\mathsf{G}}\ D = \{\}$
4. the positive literal $A$ is strictly maximal in $D$

The ground factoring rule is as follows.

$$\frac{\overbrace{A \vee A \vee D'}^{D}}{\underbrace{A \vee D'}_{C}}\ \mathsf{ground\_factoring}\ D\ C$$

Side conditions:

1. $sel_{\mathsf{G}}\ D = \{\}$
2. the positive literal $A$ is maximal in $D$

The inference rules are more general than we need as they support a selection function $sel_{\mathsf{G}}$. We actually do not need a selection function in this chapter, but kept it in the formalization of the calculus because we preferred the added generality. For the rest of this chapter, we set the selection function such that it never selects anything (i.e., $\forall C.\ sel_{\mathsf{G}}\ C = \{\}$).

We note that both inference rules are deterministic. This will play an important role in Section 7.5 where we will specify a deterministic strategy.

**Lemma 7.1.** *The inference rules* ground_resolution *and* ground_factoring *are deterministic w.r.t. their premises:*

$$\forall D\ E\ C\ C'.\ \mathsf{ground\_resolution}\ D\ E\ C \longrightarrow \mathsf{ground\_resolution}\ D\ E\ C' \longrightarrow C = C'$$

$$\forall D\ C\ C'.\ \mathsf{ground\_factoring}\ D\ C \longrightarrow \mathsf{ground\_factoring}\ D\ C' \longrightarrow C = C'$$

The proof of refutational completeness of ground ordered resolution is based on building a model of a saturated clause set. The model takes the form of an Herbrand

interpretation, i.e., a set of atoms that are considered true. We define two mutually recursive functions that construct such an atom set for a given clause set.

**Definition 7.2** (Model Construction for Ground Resolution)**.** *Let* $N^{\prec_{\mathsf{cls}} D} = \{C \in N \mid C \prec_{\mathsf{cls}} D\}$ *for any clause set* $N$ *and clause* $D$*. The mutually recursive functions* epsilon :: $'f$ *gclause set* $\Rightarrow$ $'f$ *gclause* $\Rightarrow$ $'f$ *gterm set and* interp :: $'f$ *gclause set* $\Rightarrow$ $'f$ *gterm set generate an Herbrand interpretation for a given clause set:*

$$\forall N\ C.\ \mathsf{epsilon}\ N\ C = \{A \mid C \in N \wedge sel_{\mathsf{G}}\ C = \{\} \wedge A \text{ is strictly maximal in } C \wedge \mathsf{interp}\ (N^{\prec_{\mathsf{cls}} C}) \not\models_{\mathsf{cls}} C\}$$

$$\forall N.\ \mathsf{interp}\ N = \bigcup_{C \in N} \mathsf{epsilon}\ N\ C$$

The model construction iterates over the clause set, starting from the least clause following the ordering $\prec_{\mathsf{cls}}$, and collects a set of atoms considered true. At each iteration, epsilon returns a set of atoms that are added to the interpretation: either the considered clause is already true w.r.t. the interpretation, in which case epsilon returns the empty set, or epsilon returns a single atom that makes the clause true. We say that a clause $C$ is *productive* w.r.t. $N$ if epsilon $N\ C \neq \{\}$. Note that the produced atom is unique, i.e., $\forall N\ C.\ |\mathsf{epsilon}\ N\ C| \leq 1$, because the strictly maximal element in a clause is unique w.r.t. the total ordering $\prec_{\mathsf{cls}}$.

### 7.2.2 Simulations between Transition Systems

The simulation framework described in Chapter 6 was initially developed to prove the correctness of bytecode optimizations described in Chapter 8. It was however developed with generality in mind, hoping that it could be reused in other projects involving simulation proofs. This formalization of simulation between ordered resolution and SCL(FOL) was the perfect opportunity to apply the framework in a different context.

The framework turned out to be general enough to be reused without any modification for our use case of formalizing the simulations between different logical calculi. For the sake of self-completeness, we will repeat here the main definitions needed in this chapter and refer the reader to Chapter 6 for more details on the framework and the results it provides.

**Definition 7.3** (Transition System)**.** *A* transition system $\langle \mathcal{R}; \mathcal{F} \rangle$ *is composed of a transition relation* $\mathcal{R}$ :: $'a \Rightarrow 'a \Rightarrow$ *bool between states of type* $'a$ *and a predicate* $\mathcal{F}$ :: $'a \Rightarrow$ *bool identifying final states. A final, or accepting, state is a state that can be interpreted as a valid result of the computation expressed by the transition system.*

A transition system corresponds to an interpretation of the locale semantics of Section 6.3.

Some states of a transition system might be impossible to leave once reached: we say of such a state that it is stuck. A state is not "bad" because it is stuck; it only means that it cannot perform any transition to another state.

**Definition 7.4** (Stuck State)**.** *Let* $\mathcal{R}$ :: $'a \Rightarrow 'a \Rightarrow$ *bool be a transition relation. Let* $x$ :: $'a$ *be a state. We say that* $x$ *is* stuck *if there is no possible transition starting from it:* $\nexists x'.\ \mathcal{R}\ x\ x'$.

There is a subtle distinction between final states, which represent a valid result of a computation, and stuck states, which represent the end of a computation. A state that is neither final nor stuck represents some intermediate state in an ongoing computation. A state that is final and not stuck represents a valid result in an ongoing computation. A state that is final and stuck represents a valid result of a completed computation. A state that is not final and stuck represents some kind of error state; the computation was completed without any valid result.

**Example 7.5.** *The type char stream represents an infinite stream of characters, a.k.a. an infinite word. The functions* shd :: *char stream* ⇒ *char and* stl :: *char stream* ⇒ *char stream return respectively the head (i.e., the first element) and the tail (i.e., all but the first element) of a stream.*

*Let* A, B *and* C *be the only distinct elements of type flag. A state* $\langle f ; w \rangle$ *consists of a flag* $f$ *and an infinite word* $w$*. Let* $\mathcal{R}$ :: *flag* × *char stream* ⇒ *flag* × *char stream* ⇒ *bool be a transition relation. Let* $\mathcal{F}$ :: *flag* × *char stream* ⇒ *bool be a predicate identifying final states. The transition system* $\langle \mathcal{R}; \mathcal{F} \rangle$ *reads characters from an infinite word one at a time and accepts any sequence of repeating "ab".*

*The transition relation is specified as follows.*

$\mathcal{R}\ \langle \mathsf{A}; w \rangle\ \langle \mathsf{B}; \mathsf{stl}\ w \rangle$
*Side conditions: 1.* $\mathsf{shd}\ w =$ *"a"*

$\mathcal{R}\ \langle \mathsf{A}; w \rangle\ \langle \mathsf{C}; w \rangle$
*Side conditions: 1.* $\mathsf{shd}\ w \neq$ *"a"*

$\mathcal{R}\ \langle \mathsf{B}; w \rangle\ \langle \mathsf{A}; \mathsf{stl}\ w \rangle$
*Side conditions: 1.* $\mathsf{shd}\ w =$ *"b"*

$\mathcal{R}\ \langle \mathsf{B}; w \rangle\ \langle \mathsf{C}; w \rangle$
*Side conditions: 1.* $\mathsf{shd}\ w \neq$ *"b"*

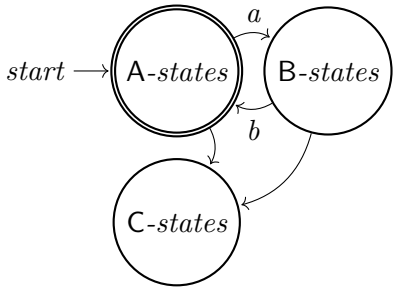
*The predicate identifying final states is specified as follows.*

$$\forall s.\ \mathcal{F}\ s \longleftrightarrow (\exists w.\ s = \langle \mathsf{A}; w \rangle)$$

*The flag* A *identifies states that expect an "a" character to start a new "ab" sequence; let us call these* A*-states. The flag* B *identifies states that expect a "b" character to end an "ab" sequence; let us call these* B*-states. The flag* C *identifies erroneous states that do not not expect any more character; let us call these* C*-states. The two first rules of* $\mathcal{R}$ *specify that an* A*-state transitions to a* B*-state upon reading an "a" and to a* C*-state otherwise. The two last rules of* $\mathcal{R}$ *specify that a* B*-state transitions to an* A*-state upon reading an "b" and to a* C*-state otherwise. The* A*-states are final, or accepting, and can be interpreted as a successful read of a sequence of repeating "ab" characters. The* C*-states are stuck, because there is no possible transition starting from them, and can be interpreted as erroneous states.*

*To analyze an infinite word* $w$*, we would start a run of the transition system from the initial state* $\langle \mathsf{A}; w \rangle$*.*

*A visual representation of this transition system follows.*




Sometimes, two transition systems are expressed differently but represent the same computation. This relationship can be expressed formally by a simulation. Intuitively, we say that a transition system $\langle \mathcal{R}_b; \mathcal{F}_b \rangle$ simulates another transition system $\langle \mathcal{R}_a; \mathcal{F}_a \rangle$ if $\langle \mathcal{R}_b; \mathcal{F}_b \rangle$ is able to do everything that $\langle \mathcal{R}_a; \mathcal{F}_a \rangle$ can do.

**Definition 7.6** (Forward Simulation and Backward Simulation, Bisimulation)**.** *Let* $\mathcal{R}_a :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{R}_b :: 'b \Rightarrow 'b \Rightarrow bool$ *be two transition relations. Let* $\sim :: 'i \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ *be an annotated matching relation between states of types* $'a$ *and* $'b$ *with an annotation of type* $'i$*. Let* $< :: 'i \Rightarrow 'i \Rightarrow bool$ *be a well-founded relation on annotations. We say that* $\mathcal{R}_b$ simulates $\mathcal{R}_a$ *w.r.t.* $\sim$ *and* $<$ *if, following a step of* $\mathcal{R}_a$*,* $\mathcal{R}_b$ *either progresses or stutters finitely often:*

$$\forall i\ a\ b.\ a \overset{i}{\sim} b \longrightarrow \mathcal{R}_a\ a\ a' \longrightarrow (\exists b'\ i'.\ \mathcal{R}_b^+\ b\ b' \wedge a' \overset{i'}{\sim} b') \vee (\exists i'.\ a' \overset{i'}{\sim} b \wedge i' < i)$$

*Let* $\prec :: 'i \Rightarrow 'i \Rightarrow bool$ *and* $\sqsubset :: 'i \Rightarrow 'i \Rightarrow bool$ *be two well-founded relations on annotations. We say that there is a* backward simulation *between* $\mathcal{R}_a$ *and* $\mathcal{R}_b$ *if* $\mathcal{R}_b$ *simulates* $\mathcal{R}_a$ *w.r.t.* $\sim$ *and* $\prec$*. We say that there is a* forward simulation *between* $\mathcal{R}_a$ *and* $\mathcal{R}_b$ *if* $\mathcal{R}_a$ *simulates* $\mathcal{R}_b$ *w.r.t.* $(\lambda i\ y\ x.\ x \overset{i}{\sim} y)$ *and* $\sqsubset$*. The* $\lambda$*-abstraction swaps the order of the arguments so that values of type* $'a$ *are on the left of* $\sim$ *and values of type* $'b$ *are on the right. We say that there is a* bisimulation *between* $\mathcal{R}_a$ *and* $\mathcal{R}_b$ *w.r.t.* $\sim$*,* $\prec$*, and* $\sqsubset$ *if there is both a forward simulation w.r.t.* $\sim$ *and* $\prec$ *and a backward simulation w.r.t.* $\sim$ *and* $\sqsubset$*. Note that the matching relation must be the same but that the well-founded relations may be different.*

We might omit to explicitly specify the matching relation or the well-founded relations if they are either clear from the context or not relevant to the discussion.

A forward simulation, backward simulation, and bisimulation corresponds respectively to an interpretation of the locale forward_simulation, backward_simulation, and bisimulation of Section 6.3.

## 7.3 Proof Outline

Figure 7.1 presents the calculi we formalized and the simulations we proved. At the very top, ORD-RES is a calculus that nondeterministically applies the inference rules of ground ordered resolution described in Section 7.2.1 until a contradiction is derived or the clause set is saturated. At the very bottom, SCL(FOL) corresponds to regular SCL as described in Section 5.3.

There are multiple challenges to proving the equivalence of ORD-RES and SCL(FOL):

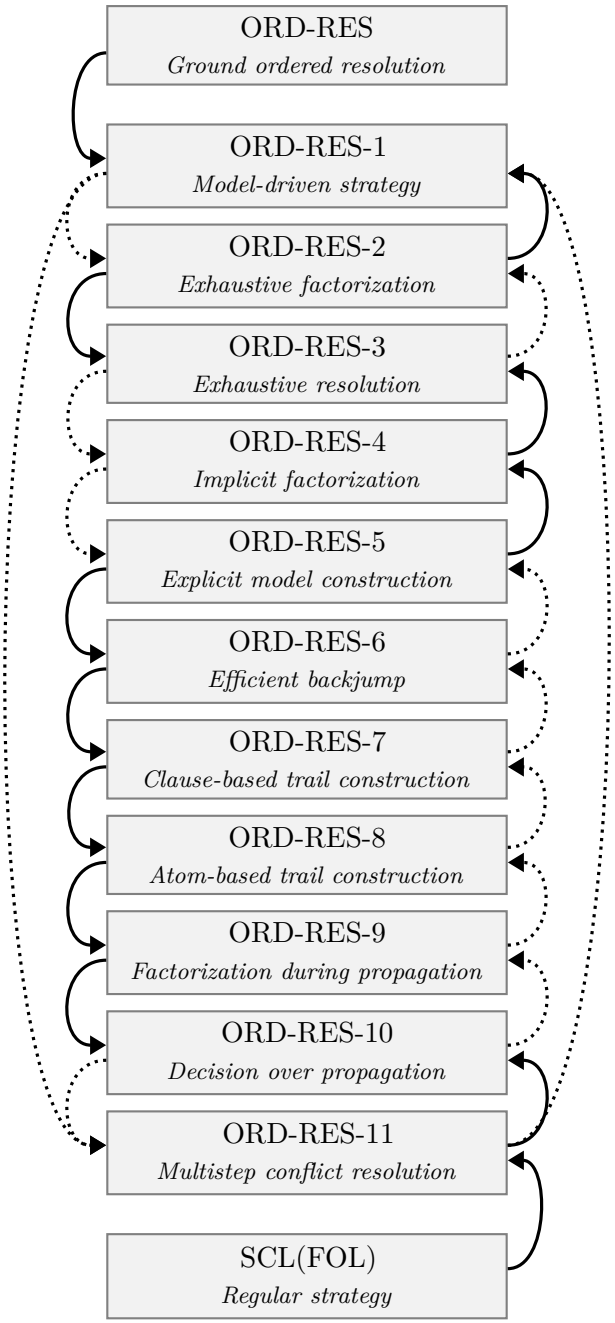



Figure 7.1: The calculi and proven simulations. An arrow from a calculus $\mathcal{A}$ to a calculus $\mathcal{B}$ means that $\mathcal{A}$ simulates $\mathcal{B}$ (i.e., if $\mathcal{B}$ makes a transition, then $\mathcal{A}$ eventually also makes some corresponding transitions); thus, a down-facing arrow corresponds to a backward simulation and an up-facing arrow corresponds to a forward simulation. A bisimulation consists of both an up- and a down-facing arrow. A solid arrow was proven manually by case analysis. A dotted arrow was proven semi-automatically using either Lemma 7.7 or composition of simulations.

1. both calculi are nondeterministic
2. ORD-RES can learn redundant clauses whereas SCL(FOL) was specifically designed to only learn nonredundant clauses
3. ORD-RES is not explicitly tied to a model construction whereas SCL(FOL) explicitly and iteratively builds a partial model
4. ORD-RES assumes a fixed ordering on atoms whereas SCL(FOL) uses a dynamic ordering based on the constructed partial model
5. ORD-RES performs a single resolution or factorization steps per inferred clause whereas SCL(FOL) typically performs multiple resolution and factoring steps per inferred clause.

Bromberger et al. were the first to face and solve these challenges in their work, but their solution is difficult to formalize. Therefore, we developed an alternative solution optimized for formalization.

Following Bromberger et al., our solution to challenges 1 and 2 is to define the ORD-RES-1 calculus, which uses the model construction found in the proof of refutational completeness of ground ordered resolution, described in Section 7.2.1, to guide the inference steps. This calculus ORD-RES-1 corresponds exactly to their strategy SUP-MO. The first theorem we prove shows that ORD-RES-1 is a strategy for ORD-RES; a strategy takes the form of a transition systems whose transitions are a subset of those from the base calculus.

For challenges 3, 4, and 5, Bromberger et al. specified a single strategy SCL-SUP. In contrast, our solution is to perform ten smaller and more manageable refinement steps that culminate in the ORD-RES-11 calculus. This calculus ORD-RES-11, although it is defined very differently from SCL-SUP, performs the same inferences in the same order.

The ten refinement steps share a common structure. For each refinement step $i$, starting from the already defined calculus ORD-RES-$i$, we define a new calculus ORD-RES-$(i+1)$ with small changes (e.g., to the data structures) or by merging or splitting steps. We define an appropriate matching relation between states of the two calculi. We prove basic properties of the calculi and also either a forward or a backward simulation. We then use these to discharge the assumptions of a lifting lemma to get a proof of bisimulation between ORD-RES-$i$ and ORD-RES-$(i+1)$. This results in a chain of small bisimulations.

We then prove our second theorem, which states that there exists a bisimulation between ORD-RES-1 and ORD-RES-11, by composing the small bisimulations of the refinement steps. This means that the two calculi are equivalent and will both either find the same model, derive the same contradiction, or never terminate.

Finally, we prove our third theorem, which states that ORD-RES-11 is a strategy for regular SCL(FOL). Here, we generalize the concept of strategy to allow projecting the enriched states of ORD-RES-11 to states of SCL(FOL). We generalize the nonsubsumption and termination theorems for SCL(FOL) [25] to this new concept of projectable strategy. By instantiating these theorems, we show that ORD-RES-11 only learns nonsubsuming clauses and always terminates. This also hold for ORD-RES-1 by bisimulation. Other results such as nonredundancy of learned clauses could also be instantiated in a similar way.

## 7.4 Lifting a Simulation to a Bisimulation

Section 7.6 presents a total of 11 bisimulations, which would naïvely require 22 simulation proofs (11 forward and 11 backward): an ambitious endeavor in both size and complexity. This is especially true when we have to stick to the same matching relation for both directions, as this restricts the ways we can specify matching states. It is however possible to substantially lessen this burden.

A folklore lemma states that it is possible to reverse a simulation between two deterministic transition systems. For example, assuming we have a forward simulation between deterministic transition systems $\mathcal{Q}$ and $\mathcal{R}$ w.r.t. the matching relation $\sim$, the folklore lemma states that there exists a matching relation $\approx$ and a backward simulation between $\mathcal{Q}$ and $\mathcal{R}$ w.r.t. $\approx$. We would end up with a forward simulation w.r.t. $\sim$ and a backward simulation w.r.t. $\approx$. This is, however, not enough to prove a bisimulation, which requires both the forward and backward simulation to be w.r.t. the same matching relation. To solve this problem, we need to extend the folklore lemma.

We prove a new lemma that states the existence of a new matching relation w.r.t. which the simulation is lifted to a bisimulation under some assumptions on the transition relation. This new matching relation works by counting the steps of both transition systems between matching states and is defined specifically to allow proving both a reversed simulation, as in the folklore lemma, but also a simulation in the same direction as the input. This result should not surprise any expert, but we nevertheless could not find it explicitly stated, let alone rigorously proven, in the literature.

This lifting lemma not only allows us to do only half of the work, it also allows us to chose the direction to prove: it gives us a bisimulation whether we provide a forward or a backward simulation. This is especially useful because one of the directions is usually much easier to prove than the other, but which one it is varies depending on the exact two transition systems considered.

**Lemma 7.7** (Lifting a Simulation to a Bisimulation)**.** *Let* $\mathcal{R}_a :: {'a} \Rightarrow {'a} \Rightarrow \mathit{bool}$ *and* $\mathcal{R}_b :: {'b} \Rightarrow {'b} \Rightarrow \mathit{bool}$ *be transition relations. Let* $\sim :: {'i} \Rightarrow {'a} \Rightarrow {'b} \Rightarrow \mathit{bool}$ *be an annotated matching relation between states of types* $'a$ *and* $'b$ *with an annotation of type* $'i$*. Let* $< :: {'i} \Rightarrow {'i} \Rightarrow \mathit{bool}$ *be a well-founded relation on annotations. If*

- $\mathcal{R}_a$ *is right-unique (i.e., functional):*
  $\forall x\ y\ z.\ \mathcal{R}_a\ x\ y \longrightarrow \mathcal{R}_a\ x\ z \longrightarrow y = z$
- $\mathcal{R}_b$ *is right-unique:*
  $\forall x\ y\ z.\ \mathcal{R}_b\ x\ y \longrightarrow \mathcal{R}_b\ x\ z \longrightarrow y = z$
- *matching states w.r.t.* $\sim$ *agree on whether they are stuck or not:*
  $\forall i\ a\ b.\ a \overset{i}{\sim} b \longrightarrow (a \text{ is stuck} \longleftrightarrow b \text{ is stuck})$
- $\mathcal{R}_b$ *simulates* $\mathcal{R}_a$ *w.r.t.* $\sim$ *and* $<$

*then there exists a matching relation* $\approx :: \mathbb{N} \times \mathbb{N} \Rightarrow {'a} \Rightarrow {'b} \Rightarrow \mathit{bool}$ *and a well-founded relation* $\ll :: \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \times \mathbb{N} \Rightarrow \mathit{bool}$ *such that the following holds:*

- *states matching w.r.t.* $\sim$ *also match w.r.t.* $\approx$*:*
  $\forall i\ a\ b.\ a \overset{i}{\sim} b \longrightarrow (\exists j.\ a \overset{j}{\approx} b)$
- *states matching w.r.t.* $\approx$ *either are both stuck and matching w.r.t.* $\sim$ *or can*

*both progress to states matching w.r.t.* $\sim$*:*
$\forall j\ a\ b.\ a \overset{j}{\approx} b \longrightarrow$
$(a$ *is stuck* $\wedge\ b$ *is stuck* $\wedge\ (\exists i.\ a \overset{i}{\sim} b)) \vee$
$(\exists a'\ b'.\ \mathcal{R}_a^+\ a\ a' \wedge \mathcal{R}_b^+\ b\ b' \wedge (\exists i.\ a' \overset{i}{\sim} b'))$

- *there is a bisimulation between* $\mathcal{R}_a$ *and* $\mathcal{R}_b$ *w.r.t.* $\approx$ *and* $\ll$

The third hypothesis of Lemma 7.7, i.e., that matching states agree on stuck states, is cumbersome to discharge in practice. The proof usually consists of proving both direction of the iff separately; each direction requires a long and complex case analysis of the possible transition rules of the transitions systems.

To avoid this difficulty, we introduce the concepts of safe states and well-behaved transition systems. We then use these new concepts to more easily discharge the hypothesis that matching states agree on stuck states.

**Definition 7.8** (Safe State)**.** *Let* $\langle \mathcal{R}; \mathcal{F} \rangle$ *be a transition system with* $\mathcal{R} :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{F} :: 'a \Rightarrow bool$*. Let* $x :: 'a$ *be a state. We say that* $x$ *is* safe *if all stuck states reachable from it are final:* $\forall y.\ \mathcal{R}^*\ x\ y \longrightarrow (y$ *is stuck*$) \longrightarrow \mathcal{F}\ y$*.*

**Definition 7.9** (Wellbehaved Transition System)**.** *Let* $\langle \mathcal{R}; \mathcal{F} \rangle$ *be a transition system with* $\mathcal{R} :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{F} :: 'a \Rightarrow bool$*. Let* $\mathcal{S} :: 'a \Rightarrow bool$ *be a predicate identifying safe states. We say that* $\langle \mathcal{R}; \mathcal{F} \rangle$ *is* well-behaved *w.r.t.* $\mathcal{S}$ *if it fulfills the following conditions:*

- $\mathcal{R}$ *is right-unique:*
  $\forall x\ y\ z.\ \mathcal{R}\ x\ y \longrightarrow \mathcal{R}\ x\ z \longrightarrow y = z$
- *final states are stuck:*
  $\forall x.\ \mathcal{F}\ x \longrightarrow (x$ *is stuck*$)$
- *states fulfilling* $\mathcal{S}$ *are safe:*
  $\forall x.\ \mathcal{S}\ x \longrightarrow (x$ *is safe*$)$

*If all states are safe, we omit the predicate* $\mathcal{S}$ *and simply say that* $\langle \mathcal{R}; \mathcal{F} \rangle$ *is* well-behaved*.*

Definition 7.9 was chosen so that final states of well-behaved transition systems coincide with stuck states.

**Lemma 7.10.** *Let* $\langle \mathcal{R}; \mathcal{F} \rangle$ *be a transition system with* $\mathcal{R} :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{F} :: 'a \Rightarrow bool$*. Let* $\mathcal{S} :: 'a \Rightarrow bool$ *be a predicate identifying safe states. If* $\langle \mathcal{R}; \mathcal{F} \rangle$ *is well-behaved w.r.t.* $\mathcal{S}$*, then its final states under* $\mathcal{S}$ *are exactly the stuck states:* $\forall x.\ \mathcal{S}\ x \longrightarrow (\mathcal{F}\ x \longleftrightarrow x$ *is stuck*$)$*.*

We can use Lemma 7.10 to lift agreement on final states to agreement on stuck states.

**Lemma 7.11.** *Let* $\langle \mathcal{R}_a; \mathcal{F}_a \rangle$ *be a transition system with* $\mathcal{R}_a :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{F}_a :: 'a \Rightarrow bool$*. Let* $\langle \mathcal{R}_b; \mathcal{F}_b \rangle$ *be a transition system with* $\mathcal{R}_b :: 'b \Rightarrow 'b \Rightarrow bool$ *and* $\mathcal{F}_b :: 'b \Rightarrow bool$*. Let* $\sim :: 'i \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ *be an annotated matching relation between states of types* $'a$ *and* $'b$ *with an annotation of type* $'i$*. If*

- $\langle \mathcal{R}_a; \mathcal{F}_a \rangle$ *is well-behaved w.r.t.* $\lambda a.\ \exists i\ b.\ a \overset{i}{\sim} b$
- $\langle \mathcal{R}_b; \mathcal{F}_b \rangle$ *is well-behaved w.r.t.* $\lambda b.\ \exists i\ a.\ a \overset{i}{\sim} b$

- *matching states w.r.t.* $\sim$ *agree on whether they are final or not:*
  $\forall i\ a\ b.\ a \overset{i}{\sim} b \longrightarrow (\mathcal{F}_a\ a \longleftrightarrow \mathcal{F}_b\ b)$

*then matching states w.r.t.* $\sim$ *agree on whether they are stuck or not:*
$\forall i\ a\ b.\ a \overset{i}{\sim} b \longrightarrow (a \textit{ is stuck} \longleftrightarrow b \textit{ is stuck})$.

Combining Lemmas 7.7 and 7.11, we only need to show that two transitions systems are well-behaved and agree on final states to lift a simulation to a bisimulation. This is advantageous because it is usually easier to prove that two transition systems agree on final states than to prove that they agree on stuck states.

## 7.5 A Strategy for Ground Ordered Resolution

**ORD-RES** is a standard ground resolution calculus that nondeterministically applies the inference rules presented in Section 7.2.1. The calculus is defined as a transition system $\langle \Rightarrow_{\mathsf{ORD\text{-}RES}}; \mathsf{final}_{\mathsf{ORD\text{-}RES}} \rangle$ on states of type $'f\ state_{ord\text{-}res} = 'f\ gclause\ fset$; the finite set contains both initial and learned clauses. We first present the definition of final states because this notion is used in the definition of the transition relation.

**Definition 7.12** (ORD-RES Final States)**.** *A state is considered final by* $\mathsf{final}_{\mathsf{ORD\text{-}RES}}$ ::

$'f\ state_{ord\text{-}res} \Rightarrow bool$ *if either it contains the empty clause or model construction succeeds (i.e., the constructed model makes all clauses true):*

$$\forall N.\ \mathsf{final}_{\mathsf{ORD\text{-}RES}}\ N \longleftrightarrow (\bot \in N \vee (\forall C \in N.\ \mathsf{interp}\ N^{\prec_{\mathsf{cls}} C} \models C))$$

**Definition 7.13** (ORD-RES Transition Relation)**.** *The transition relation* $\Rightarrow_{\mathsf{ORD\text{-}RES}}$::

$'f\ state_{ord\text{-}res} \Rightarrow 'f\ state_{ord\text{-}res} \Rightarrow bool$ *is specified as follows.*

***Factoring*** $N \Rightarrow_{\mathsf{ORD\text{-}RES}} (N \cup \{C'\})$
*Side conditions:*

1. $\neg\mathsf{final}_{\mathsf{ORD\text{-}RES}}\ N$
2. $C \in N$
3. $C'$ *is inferred by factorization of* $C$

***Resolution*** $N \Rightarrow_{\mathsf{ORD\text{-}RES}} (N \cup \{E\})$
*Side conditions:*

1. $\neg\mathsf{final}_{\mathsf{ORD\text{-}RES}}\ N$
2. $C \in N$
3. $D \in N$
4. $E$ *is inferred by resolution of* $C$ *and* $D$

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES}}$ consists of two rules: the first rule learns a new clause by factorization of a nondeterministically chosen clause and the second rule learns a new clause by resolution of two nondeterministically chosen clauses.

The termination condition of $\Rightarrow_{\mathsf{ORD\text{-}RES}}$ (Condition 1 of both rules) is nonstandard; ordered resolution usually terminates when all possible inferences are redundant. We

deviate from the standard termination condition because the simulation framework (Chapter 6) requires the considered calculi to agree on final states and because SCL(FOL) (Chapter 5), which we ultimately want to relate to ORD-RES through simulation, terminates as soon as it derives a contradiction or finds a partial model that makes all clauses true. So we defined $\mathsf{final}_{\text{ORD-RES}}$ to closely resemble the corresponding predicate of SCL(FOL) and use a nonstandard termination condition in $\Rightarrow_{\text{ORD-RES}}$. This change is justified by the following observations.

- Once the empty clause is derived, any following inference would be redundant.
- Once model construction succeeds, it keeps succeeding even if more inferences are performed.

**ORD-RES-1** is a deterministic calculus that uses the model construction found in the proof of refutational completeness to guide the order of the inference rules. The calculus is defined as a transition system $\langle \Rightarrow_{\text{ORD-RES-1}}; \mathsf{final}_{\text{ORD-RES-1}} \rangle$ on states of type $'f\ state_{ord\text{-}res\text{-}1} = 'f\ state_{ord\text{-}res}$; the type of states is the exact same as in ORD-RES.

**Definition 7.14** (Least False Clause)**.** *Let $N :: 'f\ gclause\ fset$ be a finite set of ground clauses. Let $C :: 'f\ gclause$ be a ground clause. We say that $C$ is the* least false clause *in $N$ if it is the least clause not made true by the model construction:*

$$C \text{ is the least clause in } \{D \in N \mid \mathsf{interp}\ N^{\prec_{\mathsf{cls}} D} \not\models D\}$$

Note that the empty clause $\bot$ is always the least false clause in any set it is a member of, because $\bot$ is always false, i.e., $\forall \mathcal{I}.\ \mathcal{I} \not\models \bot$, and $\bot$ is the least of all clause, i.e., $\forall C.\ \bot \preceq_{\mathsf{cls}} C$.

**Definition 7.15** (ORD-RES-1 Transition Relation)**.** *The transition relation $\Rightarrow_{\text{ORD-RES-1}}$::*

*$'f\ state_{ord\text{-}res\text{-}1} \Rightarrow 'f\ state_{ord\text{-}res\text{-}1} \Rightarrow bool$ is specified as follows.*

***Factoring*** $N \Rightarrow_{\text{ORD-RES-1}} (N \cup \{D'\})$
*Side conditions:*

1. *$D$ is the least false clause in $N$*
2. *$L$ is a maximal literal in $D$*
3. *$L$ is a positive literal*
4. *$D'$ is inferred by factorization of $D$*

***Resolution*** $N \Rightarrow_{\text{ORD-RES-1}} (N \cup \{D'\})$
*Side conditions:*

1. *$D$ is the least false clause in $N$*
2. *$L$ is a maximal literal in $D$*
3. *$L$ is a negative literal*
4. *$C \in N$*
5. *$C$ produces the atom of $L$ w.r.t. $N$*
6. *$D'$ is inferred by resolution of $C$ and $D$*

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}}$ consists again of two rules for factorization and resolution, but the side conditions differ substantially from those found in $\Rightarrow_{\mathsf{ORD\text{-}RES}}$. Condition 1 of $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}}$ imply conditions 1 and 2 of $\Rightarrow_{\mathsf{ORD\text{-}RES}}$ but also deterministically specify the clause to use. Conditions 2 and 3 of $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}}$ deterministically specify whether to use the factorization or resolution rule. Condition 5 of the resolution rule deterministically specifies the second clause to resolve with.

**Definition 7.16** (ORD-RES-1 Final States)**.** *A state is considered final by* $\mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}1}} :: {'f}\ state_{ord\text{-}res\text{-}1} \Rightarrow bool$ *if it is considered final by* $\mathsf{final}_{\mathsf{ORD\text{-}RES}}$*:*

$$\forall N.\ \mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}1}}\ N \longleftrightarrow \mathsf{final}_{\mathsf{ORD\text{-}RES}}\ N$$

Now that ORD-RES-1 is fully specified, we proceed to prove that it is well-behaved.

**Lemma 7.17.** *The transition system* $\langle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}}; \mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}1}} \rangle$ *is well-behaved.*

*Proof Sketch.* By definition of well-behaved transition systems, we must prove the following:

- $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}}$ is right-unique:
  $\forall x\ y\ z.\ x \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}} y \longrightarrow x \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}} z \longrightarrow y = z$
- final states are stuck:
  $\forall x.\ \mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}1}}\ x \longrightarrow (x \text{ is stuck})$
- all states are safe:
  $\forall x.\ (x \text{ is safe})$

□

We then prove our first theorem stating that ORD-RES-1 is a strategy for ORD-RES (i.e., transitions in ORD-RES-1 are a subset of transitions in ORD-RES).

**Theorem 7.18** (ORD-RES-1 Strategy for ORD-RES)**.** *Let* $N$ *and* $N'$ *be finite sets of ground clauses. If* $N \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}1}} N'$*, then* $N \Rightarrow_{\mathsf{ORD\text{-}RES}} N'$*.*

Note that ORD-RES-1 is not only a strategy of ORD-RES, but that it is also refutationally complete. This is justified by the following observations.

- The transition rules reuse the factoring and resolution inferences from Section 7.2.1, which we proved refutationally complete.
- The termination condition does not impact refutational completeness.
- Our choice of the clauses on which to apply the inferences is guided by the model construction from Section 7.2.1, which is used in the proof of refutational completeness of ordered resolution; this means that ORD-RES-1 performs exactly the required inferences for refutational completeness.

## 7.6 Refinement Steps

Our first refinement step will bring us closer to SCL(FOL) by performing multiple factorization steps at once.

**ORD-RES-2** performs exhaustive factorization in a single step. The calculus is defined as a transition system $\langle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}2}}; \mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}2}} \rangle$ on states of type $'f\,state_{ord\text{-}res\text{-}2} = 'f\,gclause\,fset \times 'f\,gclause\,fset \times 'f\,gclause\,fset$. A state of ORD-RES-2 is a tuple $\langle N; U_r; U_f \rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses inferred by resolution, and a finite set $U_f$ of clauses inferred by factorization.

Before defining the transition relation and predicate on final states, we define a function to perform exhaustive factorization.

**Definition 7.19** (Exhaustive Factorization)**.** *Let $C :: 'f\,gclause$ be a ground clause. The function* efac *$:: 'f\,gclause \Rightarrow 'f\,gclause$ performs exhaustive factorization of its argument $C$ through repeated application of the factorization rule:*

$$\begin{aligned}\mathsf{efac}\ C = (\mathsf{THE}\ C'.\ \mathsf{ord\_res.ground\_factoring}^*\ C\ C' \wedge \\ (\nexists C''.\ \mathsf{ord\_res.ground\_factoring}\ C'\ C''))\end{aligned}$$

The Isabelle/HOL syntax $\mathsf{THE}\,x.\ P\ x$ corresponds to the definite description operator and returns the one and only value $x$ for which the predicate $P$ holds.

In Definition 7.19, the first conjunct ensures that there is at least one value—by reflexivity if ord_res.ground_factoring is not applicable—and the second conjunct ensures that there is at most one value—by determinism of ord_res.ground_factoring there is only one transitive run and only the very last element of this run is selected.

**Definition 7.20** (ORD-RES-2 Transition Relation)**.** *The transition relation* $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}2}}$*::*

*$'f\,state_{ord\text{-}res\text{-}2} \Rightarrow 'f\,state_{ord\text{-}res\text{-}2} \Rightarrow bool$ is specified as follows.*

***Factoring*** $\langle N; U_r; U_f \rangle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}2}} \langle N; U_r; U_f \cup \{\mathsf{efac}\ D\} \rangle$
*Side conditions:*

1. *$D$ is the least false clause in $N \cup U_r \cup U_f$*
2. *$L$ is a maximal literal in $D$*
3. *$L$ is a positive literal*

***Resolution*** $\langle N; U_r; U_f \rangle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}2}} \langle N; U_r \cup \{D'\}; U_f \rangle$
*Side conditions:*

1. *$D$ is the least false clause in $N \cup U_r \cup U_f$*
2. *$L$ is a maximal literal in $D$*
3. *$L$ is a negative literal*
4. *$C \in N \cup U_r \cup U_f$*
5. *$C$ produces the atom of $L$ w.r.t. $N \cup U_r \cup U_f$*
6. *$D'$ is inferred by resolution of $C$ and $D$*

**Remark 7.21.** *The first side condition of the factorization rule searches for the least false clause in $N \cup U_r \cup U_f$ but it would sufficient to only search in the clauses $N \cup U_r$. The finite set $U_f$ contains exhaustively factorized clauses, which implies that they all have a strictly maximal positive literal. From this and Definition 7.2 follows that these clauses will always be either already true, or made true by* epsilon*. This means that the clauses in $U_f$ will never be false and, thus, never selected as the*

*least false clause. We nonetheless chose to use $N \cup U_r \cup U_f$ in the factorization rule because it allows us to reuse the predicate from ORD-RES-1 corresponding to this side condition instead of defining a new one. This choice is also more consistent with the side condition of the resolution rule: we can more easily see that the two first side conditions of both the factorization and resolution rules are the same.*

**Definition 7.22** (ORD-RES-2 Final States)**.** *A state is considered final by* $\mathsf{final}_{\text{ORD-RES-2}}$ :: *'f state$_{ord\text{-}res\text{-}2}$ ⇒ bool if the union of its clause sets is considered final by* $\mathsf{final}_{\text{ORD-RES}}$*:*

$$\forall N\ U_r\ U_f.\ \mathsf{final}_{\text{ORD-RES-2}}\ \langle N; U_r; U_f\rangle \longleftrightarrow \mathsf{final}_{\text{ORD-RES}}\ (N \cup U_r \cup U_f)$$

We again start by showing that ORD-RES-2 is well-behaved.

**Lemma 7.23.** *The transition system* $\langle \Rightarrow_{\text{ORD-RES-2}}; \mathsf{final}_{\text{ORD-RES-2}}\rangle$ *is well-behaved.*

*Proof Sketch.* By definition of well-behaved transition systems, we must prove the following:

- $\Rightarrow_{\text{ORD-RES-2}}$ is right-unique:
  $\forall x\ y\ z.\ x \Rightarrow_{\text{ORD-RES-2}} y \longrightarrow x \Rightarrow_{\text{ORD-RES-2}} z \longrightarrow y = z$
- final states are stuck:
  $\forall x.\ \mathsf{final}_{\text{ORD-RES-2}}\ x \longrightarrow (x \text{ is stuck})$
- all states are safe:
  $\forall x.\ (x \text{ is safe})$

□

To show any kind of simulation between ORD-RES-1 and ORD-RES-2, we need to identify states from both calculi that we consider equivalent. We do this with the following matching relation.

**Definition 7.24** (Matching Relation between ORD-RES-1 and ORD-RES-2)**.** *Let $M$ be a state of ORD-RES-1 and $\langle N; U_r; U_f\rangle$ be a state of ORD-RES-2. Let $i$ :: 'f gclause be a measurement used to limit stuttering (i.e., one transition system progresses while the other does not). The matching relation* $\sim$:: *'f gclause ⇒ 'f state$_{ord\text{-}res\text{-}1}$ ⇒ 'f state$_{ord\text{-}res\text{-}2}$ ⇒ bool considers $M$ equivalent to $\langle N; U_r; U_f\rangle$ w.r.t. $i$ if $M$ is equal to $N \cup U_r \cup U_f$ plus all partially factorized clauses up to the least false clause:*

$$\begin{aligned}
&M \overset{i}{\sim} \langle N; U_r; U_f\rangle \longleftrightarrow\\
&\quad i = (\textit{the least false clause in } M \textit{ if there is one or } \bot \textit{ otherwise}) \wedge\\
&\quad (\exists U.\ M = N \cup U_r \cup U_f \cup U \wedge\\
&\qquad (\forall C \in U.\ C \neq \mathsf{efac}\ C \wedge\\
&\qquad\quad (\exists C_0 \in N \cup U_r \cup U_f.\ \mathsf{ord_res.ground_factoring}^{+}\ C_0\ C \wedge\\
&\qquad\qquad (\mathsf{efac}\ C \in U_f \vee C_0 \textit{ is the least false clause in } N \cup U_r \cup U_f))))
\end{aligned}$$

The measurement $i$ is used in the proof of forward simulation when ORD-RES-1 performs a factorization step, but the inferred clause is still not exhaustively factorized. The partly factorized clause becomes ORD-RES-1's new least false clause and another factorization will be performed. During all this time, ORD-RES-2 "waits" and does

not progress. Only when ORD-RES-1 performs the ultimate factorization step does ORD-RES-2 performs an exhaustive factorization in one step. Definition 7.6 requires us to provide a measurement and a well-founded relation to ensure the finiteness of this "waiting time". Because factorization produces a strict submultiset, i.e., the original clause minus one literal, we can directly use the clause getting factorized as a measurement and $\subset$ as well-founded relation. An alternative would have been to use a natural number, i.e., the number of occurrences of the maximal positive literal in the clause, and $<$ as well-founded relation.

Definition 7.24 specifies on line 3 the existence of a finite set $U$ of partially factorized clauses saved in ORD-RES-1's state (i.e., in $M$) but not in ORD-RES-2's state (i.e., in $N \cup U_r \cup U_f$). Every clause $C \in U$ can be further factorized because applying $\mathsf{efac}$ on line 4 has an effect (i.e., $C \neq \mathsf{efac}\ C$). Line 5 further specifies that every clause $C \in U$ can be inferred by repetitive factorization—$\mathsf{ord\_res.ground\_factoring}^+$ is the transitive closure of the factorization rule—of a clause $C_0$ saved in ORD-RES-2's state (i.e., in $N \cup U_r \cup U_f$). From these two conditions, we know that $C$ is a partially factorized clause. Finally, line 6 distinguishes between two possibilities: either the clause is some intermediate step of an exhaustively factorized clause known to ORD-RES-2 (i.e., $\mathsf{efac}\ C \in U_f$), or it is some intermediate step of an ongoing factorization chain of ORD-RES-1 (i.e., $C_0$ is the least false clause in $N \cup U_r \cup U_f$). In this last case, ORD-RES-2 is currently "waiting" for ORD-RES-1 and will soon perform the exhaustive factorization of $C_0$.

The lifting lemma from Section 7.4 allows us to lift a simulation to a bisimulation if we can show that the two transitions systems are well-behaved and agree on final states. Lemmas 7.17 and 7.23 already proved that ORD-RES-1 and ORD-RES-2 are well-behaved. We still need to prove that they agree on final states.

**Lemma 7.25** (Agreement of ORD-RES-1 and ORD-RES-2 on Final States)**.** *Matching states w.r.t. $\sim$ agree on whether they are final or not.*

We now only need to prove a simulation: either a forward or a backward one. In a forward simulation, we need to use the measurement to prove that ORD-RES-2 only stutters finitely often. In a backward simulation, we need to prove that one exhaustive factorization step in ORD-RES-2 corresponds to a nonempty chain of factorization steps in ORD-RES-1: this requires to prove at each step of the chain that the inferred clause is the new least false clause. We chose the forward direction because it *a priori* felt more intuitive.

**Lemma 7.26** (Simulation between ORD-RES-1 and ORD-RES-2)**.** *There exists a forward simulation between ORD-RES-1 and ORD-RES-2 w.r.t. $\sim$ and $\subset$.*

We can now use the lifting lemma to get a bisimulation.

**Lemma 7.27** (Bisimulation between ORD-RES-1 and ORD-RES-2)**.** *There exists a bisimulation between ORD-RES-1 and ORD-RES-2.*

*Proof Sketch.* By Lemmas 7.7, 7.11, 7.17, 7.23, 7.25 and 7.26. □

All remaining refinement steps follow the same pattern: (1) we define a new calculus as a transition system that changes some aspect of the previous calculus; (2) we prove that this new calculus is well-behaved, sometimes conditionally w.r.t.

a safety predicate; (3) we define a matching relation; (4) we prove that the new calculus agrees with the previous one on final states; (5) we prove either a forward or a backward simulation; (6) we lift the simulation to a bisimulation. To save space and avoid repetitive and complex definitions, we will only sketch the noteworthy elements of the remaining refinement steps. We refer the interested reader to the Isabelle/HOL formalization for the details.

The exact choice of which changes to do and in which order was highly subjective: we always considered the remaining differences to SCL(FOL) and chose some change that would bring us closer.

The next refinement step is similar and performs multiple resolution steps at once.

**ORD-RES-3** performs exhaustive resolution in a single step. Neither the state nor the predicate on final states change. We define a function eres analogue to efac and use it in the resolution rule of the transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}3}}$ to perform all resolution steps at once.

The matching relation between ORD-RES-2 and ORD-RES-3 is analogue to the one between ORD-RES-1 and ORD-RES-2. The main difference is that we do not need a measurement to avoid stuttering.

We prove a backward simulation between ORD-RES-2 and ORD-RES-3.

**Remark 7.28** (Stuttering Measurement vs Intermediate Steps)**.** *ORD-RES-2 and ORD-RES-3 are very similar as they both replace a sequence of steps by one big step. This gave us an opportunity to try both a forward and a backward simulation and compare the proof size and complexity. On paper, the two directions felt somewhat comparable, but actually formalizing the proofs in Isabelle/HOL revealed differences. Our proof of backward simulation is bigger than our proof of forward simulation and it felt much harder to get the details right, details which we glossed over on paper. Our conclusion is that measuring stuttering, when possible, can be substantially easier than proving a chain of intermediate steps.*

For the next refinement, we address the fact that factorization is handled in two quite different ways in SCL(FOL): propagating clauses are implicitly and exhaustively factorized w.r.t. the propagated literal and conflicting clauses can be factorized w.r.t. any literal. In the model-guided strategy of ORD-RES-1, factorization is only performed w.r.t. the positive maximal literals of false clauses. In other words, these false clauses are factorized w.r.t. their positive maximal literal after which the inferred clauses become productive. This rules out SCL(FOL)'s factorization of conflicting clauses, because conflict resolution always involves the resolution rule. This means that factorization must be done implicitly during literal propagation. The next refinement brings us closer to that.

**ORD-RES-4** performs implicit factorization by saving the clauses that ought to be implicitly factorized. A state of ORD-RES-4 is a tuple $\langle N; U_r; \mathcal{F} \rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses learned by resolution, and a finite set $\mathcal{F} \subseteq N \cup U_r$ of clauses that ought to be implicitly factorized.

We define the following function to perform implicit factorization.

**Definition 7.29** (Implicit Exhaustive Factorization)**.** *Let* $\mathcal{F} :: 'f\ gclause\ fset$ *be a set of ground clauses. Let* $C :: 'f\ gclause$ *be a ground clause. The function* iefac :: $'f\ gclause\ fset \Rightarrow 'f\ gclause \Rightarrow 'f\ gclause$ *performs exhaustive factorization of its argument* $C$ *through* efac *if* $C$ *is in* $\mathcal{F}$*:*

$$\mathsf{iefac}\ \mathcal{F}\ C = (\mathsf{if}\ C \in \mathcal{F}\ \mathsf{then}\ \mathsf{efac}\ C\ \mathsf{else}\ C)$$

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}4}}$ and predicate $\mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}4}}$ on final states are similar to $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}3}}$ and $\mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}3}}$ except for the following changes.

- Every time the set of known clauses is used, the relevant clauses are implicitly factorized, i.e., ORD-RES-4 uses $\{\mathsf{iefac}\ \mathcal{F}\ C \mid C \in N \cup U_r\}$ everywhere ORD-RES-3 uses $N \cup U_r \cup U_f$.
- The transition rule handling factorization extends the finite set $\mathcal{F}$ with the clause to be factorized, i.e., it remembers to implicitly factorize the clause instead of explicitly learning the factorized clause.

The matching relation between ORD-RES-3 and ORD-RES-4 requires that the invariant $\mathcal{F} \subseteq N \cup U_r$ holds for ORD-RES-4 and that the finite set $U_f$ in ORD-RES-3 contains exactly the clauses implicitly factorized in ORD-RES-4.

We prove a forward simulation between ORD-RES-3 and ORD-RES-4.

For the next refinement, we start to address the fact that SCL(FOL) explicitly and iteratively builds a partial model while our calculi so far relied on implicitly constructing a model at each application of a transition rule.

**ORD-RES-5** explicitly builds a partial model that closely matches the implicit model construction by iterating over clauses. A state of ORD-RES-5 is a tuple $\langle N; U_r; \mathcal{F}; \mathcal{M}; \mathcal{C}\rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses learned by resolution, a finite set $\mathcal{F} \subseteq N \cup U_r$ of clauses that ought to be implicitly factorized, a partial model $\mathcal{M}$ expressed as a partial function from atoms to clauses, and an optional clause $\mathcal{C}$ that identifies the next clause to be considered.

The initial state is $\langle N; \{\}; \{\}; \epsilon; \mathcal{C}\rangle$ where $\mathcal{C}$ is either the least clause in $N$, if there is one, or † otherwise.

The domain of the partial function $\mathcal{M}$ consists of all atoms considered true in the partial model built by model construction up to but excluding the clause in $\mathcal{C}$. The codomain of $\mathcal{M}$ uniquely identifies the clauses that made each of the atoms true. At all time, if $\mathcal{C} = D$ for some clause $D$, then $\mathsf{dom}\ \mathcal{M}$ corresponds exactly to the result of model construction up to but excluding $D$ (i.e., $\mathsf{dom}\ \mathcal{M} = \mathsf{interp}\ \{\mathsf{iefac}\ \mathcal{F}\ C \mid C \in N \cup U_r\}^{\prec_{\mathsf{cls}} D}$) and all clauses less than $D$ are true w.r.t. the partial model (i.e., $\forall C \in \{\mathsf{iefac}\ \mathcal{F}\ C \mid C \in N \cup U_r\}.\ C \prec_{\mathsf{cls}} D \longrightarrow \mathsf{dom}\ \mathcal{M} \models C$). If $\mathcal{C} = \dagger$, model construction is completed and all clauses are true w.r.t. the partial model.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}5}}$ starts from the initial state and reimplements the model construction as explicit rules of the calculus. Each rule works on the next clause to be considered, i.e., $\mathcal{C} = D$ for some clause $D$. If $D$ is true w.r.t. the partial model, then it is skipped and the least clause greater than $D$ is saved in the state to be considered next time. If $D$ is false w.r.t. the partial model but produces the atom $A$, then the partial model is expanded by defining the case $A \mapsto D$ in the function $\mathcal{M}$; the least clause greater than $D$ is then saved in the state to be considered next time. If $D$ is false w.r.t. the partial model and does not produce any atom, then

either the factorization or the resolution rule applies; whichever it was, the partial model is then reset to zero and the next clause to be considered reset to the least of the known clauses. The model construction then restarts from the beginning.

The predicate $\mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}5}}$ on final states considers a state final either if $\mathcal{C} = \dagger$, i.e., model construction is completed and all clauses are true, or if $\mathcal{C} = \bot$, i.e., the empty clause was derived.

The matching relation between ORD-RES-4 and ORD-RES-5 requires that invariants of ORD-RES-5 are fulfilled and that the states are the same except for the explicit partial model which exists in ORD-RES-5 but not in ORD-RES-4. To ensure that the explicit model in ORD-RES-5 corresponds to the implicit model in ORD-RES-4, the matching relation also requires that the next clause to be considered by ORD-RES-5 is the least false clause in ORD-RES-4.

We prove a forward simulation between ORD-RES-4 and ORD-RES-5. The way we chose to define the matching relation means that, for every step made by ORD-RES-4, we have to perform the same factorization or resolution step and then construct the full sequence of ORD-RES-5 steps that builds in advance the exact same partial model, up to a least false clause, that ORD-RES-4 will compute in its the next step.

The next refinement step adds an optimization that saves us from rebuilding similar partial models again and again. When ORD-RES-5 performs a factorization or resolution step, it discards its partial model and restarts from the least known clause. This is a simple and sound strategy, but an inefficient one: the next partial model that will get built is usually very similar to the previous one. Because the model construction works by iterating over clauses in order, a better strategy is to find out the clause at which the next partial model differs from the old one and reset the partial model to that common subset. We call this operation a backjump.

**ORD-RES-6** performs a backjump when learning a new clause. Neither the state nor the predicate on final states change.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}6}}$ is similar to $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}5}}$ except for the following changes.

- The rule handling factorization keeps the partial model and sets the next clause to be considered to the exhaustively factorized clause.
- The rule handling resolution is split into three rules that handle the cases when the result of resolution is $\bot$, a nonempty clause with a positive maximal literal, or a nonempty clause with a negative maximal literal. Each case varies in the partial model that can be kept or the next clause to be considered.

The matching relation between ORD-RES-5 and ORD-RES-6 requires that invariants are fulfilled and that the states are the same.

We prove a backward simulation between ORD-RES-5 and ORD-RES-6. For every step made by ORD-RES-6, we have to perform the same step in ORD-RES-5 and then construct the full sequence of ORD-RES-5 steps that builds the exact same partial model and reaches the same next-to-be-considered clause as ORD-RES-6.

At this point, we have an explicit and efficient model construction, but its representation as a partial function is very different from the trail of annotated literals used by SCL(FOL). In the next refinement step, we change the data representation of the partial model to more closely align to SCL(FOL).

**ORD-RES-7** explicitly builds a partial model as a trail of literals by iterating over clauses. A state of ORD-RES-7 is a tuple $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses learned by resolution, a finite set $\mathcal{F} \subseteq N \cup U_r$ of clauses that ought to be implicitly factorized, a partial model $\Gamma$ expressed as a sequence of annotated literals, i.e., a trail in SCL(FOL) parlance, and an optional clause $\mathcal{C}$ that identifies the next clause to be considered.

The trail contains literals to explicitly identify whether each atom should be considered true (using a positive literal) or false (using a negative literal). All atoms that were implicitly false in ORD-RES-6 due to not being in the partial model are made explicitly false in ORD-RES-7 by storing them as negative literals in the trail. The annotated literals in the trail are actually pairs of a literal and an optional clause (the annotation). If present, the annotation saves the clause that produced the atom of the literal. Because produced atoms are always considered true, the annotated literals will always be positive.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}7}}$ is similar to $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}6}}$ except for the following changes.

- ORD-RES-7 uses predicates from SCL(FOL) for true or false clauses w.r.t. a trail everywhere ORD-RES-6 uses predicates from ORD-RES for true or false clauses w.r.t. a set of positive atoms.
- ORD-RES-7 uses trail extension or trail restriction everywhere ORD-RES-6 uses function extension or function restriction on its model.
- The rules that expand the model or skip a clause are changed to also add negative literals to the trail.
- All rules ensure that, before they apply, all atoms less than the currently considered clause's greatest atom are defined in the trail.

One difficulty arises when the currently considered clause is already true w.r.t. the trail but the truth value of the atom of its greatest literal is still undefined. At this point, it is unknown whether this atom should be added as a positive or a negative literal to the model. It could be either, depending on the following clauses. One simple but incomplete solution is to skip this clause and let the case be handled later when more information is available. But this simple solution breaks when there is no next clause, because it would end the run of the calculus with an incomplete trail missing one atom, breaking one of the key invariants of the calculus. Our complete solution is to add checks and dedicated rules to recognize these corner cases; when there is no next clause, we decide that the atom is negative.

In total, $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}7}}$ has ten rules: six for model construction, one for factorization, and three for resolution. We specified 15 invariants and needed approximately 4000 lines to prove that they are preserved by $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}7}}$.

The matching relation between ORD-RES-6 and ORD-RES-7 requires that the invariants of both calculi hold, that the states only differ in their model representation, and that the two model representations are equivalent. The matching relation defines the finite set of atoms undefined in the trail of ORD-RES-7 as a decreasing measure that limits stuttering.

We prove a backward simulation between ORD-RES-6 and ORD-RES-7. For every step made by ORD-RES-7, we have to either perform the corresponding step in ORD-RES-6 or prove that the final set of atoms undefined in the trail of ORD-RES-7

strictly decreases.

At this point, we have an explicit and efficient model construction that closely resembles the one in SCL(FOL), but it is based on iterating over clauses while SCL(FOL) iterates over atoms. In the next refinement step, we change iteration pattern.

**ORD-RES-8** explicitly builds a partial model as a trail of literals by iterating over atoms. A state of ORD-RES-8 is a tuple $\langle N; U_r; \mathcal{F}; \Gamma \rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses learned by resolution, a finite set $\mathcal{F} \subseteq N \cup U_r$ of clauses that ought to be implicitly factorized, and a partial model $\Gamma$ expressed as a trail.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}8}}$ is substantially simpler than $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}7}}$: it consists of two rules for model construction, one rule for factorization, and one rule for resolution. The model-construction rules start by finding the least atom undefined in the trail. If a clause can propagate this atom, then the trail gets expanded with a positive literal annotated with the propagating clause. If no clause can propagate this atom, but a clause can be factorized w.r.t. this atom, then the factorization rule is used. Otherwise, the trail gets expanded with a negative literal. All three previous rules are conditional on no clause being false w.r.t. the trail. If there is such a false clause, then the resolution rule is applied. All special rules present in $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}7}}$ to handle corner cases disappear because there is no need to handle a next-to-be-considered clause anymore.

The predicate $\mathsf{final}_{\mathsf{ORD\text{-}RES\text{-}8}}$ on final states considers a state final either if all atoms are defined and there is no false clause, or if $\bot$ is in the known clauses.

The matching relation between ORD-RES-7 and ORD-RES-8 requires that the invariants of both calculi hold and that the states are the same except for ORD-RES-7's next-to-be-considered clause, which must be the least clause nonskipped by ORD-RES-7. The definition of least nonskipped clause is very technical and was chosen exactly to fit the simulation proof.

We prove a backward simulation between ORD-RES-7 and ORD-RES-8. For every step made by ORD-RES-8, we have to perform a corresponding step in ORD-RES-7 and then construct a sequence of ORD-RES-7 steps that jump over all clauses that are skipped because they neither lead to an expansion of the model, nor to a factorization rule, nor to a resolution rule.

The next refinement steps aims to remove the need for a distinct rule to handle factorization, because SCL(FOL) does not have such a rule and instead performs factorization implicitly when propagating.

**ORD-RES-9** takes note of the necessary implicit factorizations during propagation. Neither the state nor the predicate on final states change.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}9}}$ is similar to $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}8}}$ except for the following changes.

- There is no rule for factorization.
- When possible, the rule propagating a positive literal to the trail performs exhaustive factorization of the annotated clause and register the clause to be implicitly factorized in the future.

The matching relation between ORD-RES-8 and ORD-RES-9 requires that the states are the same.

We prove a backward simulation between ORD-RES-8 and ORD-RES-9. For every step made by ORD-RES-9, we have to perform the corresponding step in ORD-RES-8 and if ORD-RES-9 performs some factorization when propagating, we have to construct beforehand the equivalent sequence of ORD-RES-8 steps.

The next refinement step aims to minimize the number of propagations, replacing most of them by decisions of positive literal. This is desirable because these propagations could force SCL(FOL) to perform unwanted resolution steps between an annotated clause and a conflicting clause.

**ORD-RES-10** favors decisions and propagates if, and only if, a conflict is imminent. Neither the state nor the predicate on final states change.

The transition relation $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}10}}$ is similar to $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}9}}$ except for the following changes.

- The propagation rule is only applied if it will immediately produce a conflict.
- A new rule decides a positive literal for cases when propagation could have been used but for the new conflict-producing condition.

The matching relation between ORD-RES-9 and ORD-RES-10 requires that the invariants of both calculi hold, that the states are the same except for the trails, where some propagated literals in ORD-RES-9 may be replaced by decided positive literals in ORD-RES-10.

We prove a backward simulation between ORD-RES-9 and ORD-RES-10. For every step made by ORD-RES-10, we have to perform corresponding step in ORD-RES-9. The decisions of positive literals in ORD-RES-10 correspond to a propagation in ORD-RES-9.

The next and last refinement step aims to split the exhaustive resolution step into multiple rules akin to conflict resolution in SCL(FOL). We will describe this last calculus in more details than the calculi of the intermediate steps. In Section 7.7, we will then show that this last calculus is a strategy of regular SCL(FOL).

**ORD-RES-11** splits the exhaustive resolution step into multiple steps to detect a conflict, perform one resolution step at a time, skip literals from the trail, and backtrack. A state of ORD-RES-11 is a tuple $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ consisting of a finite set $N$ of initial clauses, a finite set $U_r$ of clauses learned by resolution, a finite set $\mathcal{F} \subseteq N \cup U_r$ of clauses that ought to be implicitly factorized, a partial model $\Gamma$ expressed as a trail, and an optional conflicting clause $\mathcal{C}$.

Before defining the transition relation and predicate on final states, we first introduce a few intermediate definitions to hide complex and repetitive formulas.

**Definition 7.30** (False Clause w.r.t. ORD-RES-11)**.** *Let $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ be an ORD-RES-11 state. Let $C$ be a ground clause. We say that $C$ is a* false clause *w.r.t. $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ if it is a clause from $N \cup U_r$ that is false w.r.t. the trail $\Gamma$:*

$$C \in \{\mathsf{iefac}\ \mathcal{F}\ C \mid C \in N \cup U_r\} \wedge (C \text{ is false under } \Gamma)$$

**Definition 7.31** (Propagating Clause w.r.t. ORD-RES-11)**.** *Let $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ be an ORD-RES-11 state. Let $C$ be a ground clause. Let $L$ be a ground literal. We*

*say that* $C$ *is a* propagating clause *for* $L$ *w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ *if* $C$ *is a clause from* $N \cup U_r$ *that can propagate* $L$ *w.r.t. the trail* $\Gamma$*:*

$$C \in \{\mathsf{iefac}\ \mathcal{F}\ C \mid C \in N \cup U_r\} \wedge (L \text{ is undefined in } \Gamma) \wedge$$
$$(L \text{ is a maximal literal in } C) \wedge (\text{the clause } \{K \in C \mid K \neq L\} \text{ is false under } \Gamma)$$

**Definition 7.32** (Least Undefined Atom w.r.t. ORD-RES-11)**.** *The unary functions* $\mathsf{atoms}_{\mathsf{clss}} :: 'a$ *clause fset* $\Rightarrow 'a$ *fset and* $\mathsf{atoms}_{\mathsf{trail}} :: ('a$ *literal* $\times 'b)$ *list* $\Rightarrow 'a$ *fset return the finite set of all atoms (of any type* $'a$*) occurring respectively in a clause set and in a trail of annotated (with any type* $'b$*) literals.*

*Let* $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ *be an ORD-RES-11 state. Let* $A$ *be a ground atom. We say that* $A$ *is the* least undefined atom *w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle$ *if it is the least atom from* $N \cup U_r$ *not in the trail* $\Gamma$*:*

$$A \text{ is the least atom in } \{A_2 \in \mathsf{atoms}_{\mathsf{clss}}\ (N \cup U_r) \mid \forall A_1 \in \mathsf{atoms}_{\mathsf{trail}}\ \Gamma.\ A_1 \prec_{\mathsf{t}} A_2\}$$

We can now proceed to define the ORD-RES-11 calculus.

**Definition 7.33** (ORD-RES-11 Transition Relation)**.** *The transition relation* $\Rightarrow_{\mathsf{ORD\text{-}RES\text{-}11}}::$

$'f\ state_{ORD\text{-}RES\text{-}11} \Rightarrow 'f\ state_{ORD\text{-}RES\text{-}11} \Rightarrow bool$ *is specified as follows.*

***DecideNeg*** $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}11}} \langle N; U_r; \mathcal{F}; \Gamma'; \dagger\rangle$
*Side conditions:*

1. *there is no false clause w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
2. $A$ *is the least undefined atom w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
3. *there is no propagating clause for the positive literal* $A$ *w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
4. $\Gamma' = \Gamma, \langle \neg A; \dagger\rangle$

***DecidePos*** $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}11}} \langle N; U_r; \mathcal{F}'; \Gamma'; \dagger\rangle$
*Side conditions:*

1. *there is no false clause w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
2. $A$ *is the least undefined atom w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
3. $C$ *is the least propagating clause for the positive literal* $A$ *w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
4. $\mathcal{F}' = \mathcal{F} \cup (\mathsf{if}\ (A \text{ is the greatest literal in } C)\ \mathsf{then}\ \{\}\ \mathsf{else}\ \{C\})$
5. *there is no false clause w.r.t.* $\langle N; U_r; \mathcal{F}'; \Gamma, \langle A; \dagger\rangle; \dagger\rangle$
6. $\Gamma' = \Gamma, \langle A; \dagger\rangle$

***Propagate*** $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}11}} \langle N; U_r; \mathcal{F}'; \Gamma'; \dagger\rangle$
*Side conditions:*

1. *there is no false clause w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
2. $A$ *is the least undefined atom w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
3. $C$ *is the least propagating clause for the positive literal* $A$ *w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$
4. $\mathcal{F}' = \mathcal{F} \cup (\mathsf{if}\ (A \text{ is the greatest literal in } C)\ \mathsf{then}\ \{\}\ \mathsf{else}\ \{C\})$
5. *there is a false clause w.r.t.* $\langle N; U_r; \mathcal{F}'; \Gamma, \langle A; \dagger\rangle; \dagger\rangle$
6. $\Gamma' = \Gamma, \langle A; \mathsf{efac}\ C\rangle$

***Conflict*** $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle \Rightarrow_{\text{ORD-RES-11}} \langle N; U_r; \mathcal{F}; \Gamma; D\rangle$
*Side conditions:*

1. $D$ *is the least false clause w.r.t.* $\langle N; U_r; \mathcal{F}; \Gamma; \dagger\rangle$

***Skip*** $\langle N; U_r; \mathcal{F}; \Gamma, \langle L; \_\rangle; D\rangle \Rightarrow_{\text{ORD-RES-11}} \langle N; U_r; \mathcal{F}; \Gamma; D\rangle$
*Side conditions:*

1. $\mathsf{comp}\ L \notin D$

***Resolve*** $\langle N; U_r; \mathcal{F}; \Gamma, \langle L; C\rangle; D\rangle \Rightarrow_{\text{ORD-RES-11}} \langle N; U_r; \mathcal{F}; \Gamma, \langle L; C\rangle; D'\rangle$
*Side conditions:*

1. $\mathsf{comp}\ L \in D$
2. $D' = (C - \{L\}) \cup (D - \{\mathsf{comp}\ L\})$

***Backtrack*** $\langle N; U_r; \mathcal{F}; \Gamma, \langle L; \dagger\rangle; D\rangle \Rightarrow_{\text{ORD-RES-11}} \langle N; U_r \cup \{D\}; \mathcal{F}; \Gamma; \dagger\rangle$
*Side conditions:*

1. $\mathsf{comp}\ L \in D$

**Definition 7.34** (ORD-RES-11 Final States)**.** *A state is considered final by* $\mathsf{final}_{\text{ORD-RES-11}}$ ::

*'f* $state_{ORD\text{-}RES\text{-}11} \Rightarrow bool$ *either if there is no undefined atom and no false clause w.r.t. the state, or if the trail is empty and the conflicting clause is* $\bot$*:*

$$\begin{aligned}
&\forall N\ U_r\ \mathcal{F}\ \Gamma\ \mathcal{C}.\ \mathsf{final}_{\text{ORD-RES-11}}\ \langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle \longleftrightarrow\\
&\quad (\mathcal{C} = \dagger \wedge (\textit{there is no false clause w.r.t. } \langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle) \wedge\\
&\qquad (\textit{there is no undefined atom w.r.t. } \langle N; U_r; \mathcal{F}; \Gamma; \mathcal{C}\rangle)) \vee\\
&\quad (\mathcal{C} = \bot \wedge \Gamma = \epsilon)
\end{aligned}$$

The matching relation between ORD-RES-10 and ORD-RES-11 requires that the invariants of both calculi hold and that the states are almost the same. The only allowed difference in the states are that $\bot$, if present, is in the set of learned clause in ORD-RES-10 and the conflicting clause in ORD-RES-11.

We prove a forward simulation between ORD-RES-10 and ORD-RES-11. For every step made by ORD-RES-10, we have to perform a corresponding step in ORD-RES-11. In the case of an exhaustive resolution step in ORD-RES-10, we build a sequence of conflict step, resolution steps, trail skipping steps, and backtracking step in ORD-RES-11.

This concludes our sequence of refinements and we prove our second theorem that composes all bisimulations between ORD-RES-1 and ORD-RES-11.

**Theorem 7.35** (Bisimulation between ORD-RES-1 and ORD-RES-11)**.** *There exists a bisimulation between ORD-RES-1 and ORD-RES-11.*

*Proof Sketch.* By composition of the bisimulation between ORD-RES-1 and ORD-RES-2, the bisimulation between ORD-RES-2 and ORD-RES-3, etc. □

We have now proven that the calculi ORD-RES-1 and ORD-RES-11 behave the same (i.e., they both either find the same model), derive a contradiction, or never terminate. In Section 7.7, we will improve this result by proving that ORD-RES-11 always terminates, which in turns implies that ORD-RES-1 also always terminates.

## 7.7 A Strategy for SCL(FOL)

**SCL(FOL)** corresponds to regular SCL as described in Section 5.3.

The last theorem we prove states that ORD-RES-11 is a strategy of SCL(FOL). For this, we cannot simply show that every transition of the first is a transition of the other, as we did in Theorem 7.18, because they operate on different states. The differences are

1. the tuple members are in a different order;
2. ORD-RES-11 stores the initial clauses in its state while SCL(FOL) takes it as a parameter to its transition relation;
3. the annotations in ORD-RES-11's trail differs from those in SCL(FOL)'s trail;
4. ORD-RES-11 operates on ground terms while SCL(FOL) operates on first-order terms; and
5. ORD-RES-11 stores a finite set of clauses (to implicitly factorize) that is completely absent from SCL(FOL)'s state.

We could have avoided the differences (1), (2), and (3) if some refinements steps would have been specified differently in Section 7.6—in fact, the Isabelle/HOL formalization splits the calculi in two layers such that difference (2) is not a problem at all—but that would have been at the cost of added complexity in specifications and proofs. We could have avoided difference (4) by adding a supplementary refinement step that changes the term representation, but this would not have solved difference (5). The set of clauses to be implicitly factorized is necessary to be equivalent to ORD-RES-1 and does not exists in SCL(FOL). There is no way to reconcile these two facts when considering our current conception of what a strategy is: a subset of possible transitions is simply not expressive enough.

Our solution is to generalize the concept of strategy to something we call a *projectable strategy*. For this, we introduce the following two generalizations:

1. A strategy may work on a different set of states as long as there exists a projection from the strategy's states to those of the restricted system.
2. We only consider transitions from the strategy reachable from some initial state.

Generalization (1) is self-explanatory and solves the problem we have with difference (5) seen previously. Generalization (2) was made to allow a strategy to enforce some invariants. In our concrete case, one example of such an invariant is that the set of clauses to be implicitly factorized is a subset of the initial and learned clauses.

**Definition 7.36** (Projectable Strategy)**.** *Let* $\mathcal{R}_a :: 'a \Rightarrow 'a \Rightarrow bool$ *and* $\mathcal{R}_b :: 'b \Rightarrow 'b \Rightarrow bool$ *be two transition relations. Let* $init_a :: 'a$ *and* $init_b :: 'b$ *be initial states of* $\mathcal{R}_a$ *and* $\mathcal{R}_b$ *respectively. We say that* $\mathcal{R}_a$ *is a* projectable strategy *of* $\mathcal{R}_b$ *if there exists a projection function* $\mathcal{P} :: 'a \Rightarrow 'b$ *such that the following conditions are fulfilled.*

1. $\mathcal{P}\ init_a = init_b$
2. $\forall x\ y.\ {\mathcal{R}_a}^*\ init_a\ x \longrightarrow \mathcal{R}_a\ x\ y \longrightarrow \mathcal{R}_b\ (\mathcal{P}\ x)\ (\mathcal{P}\ y)$

Using this new concept of projectable strategy, we can now prove our third and last theorem.

**Theorem 7.37** (ORD-RES-11 Strategy for SCL(FOL))**.** *Let $N$ be a finite set of ground clauses. Let $\beta$ be a ground atom. Let $S$ and $S'$ be ORD-RES-11 states. There exists projection functions $proj_{trm}$, $proj_{clss}$, and $proj_{state}$ such that the following statements hold.*

1. *the projection of ORD-RES-11's initial state is SCL(FOL)'s initial state:* $proj_{state}\ \langle N; \{\}; \{\}; \epsilon; \dagger\rangle = \langle \epsilon; \{\}; \dagger\rangle$
2. *if*
   - *$\beta$ is greater than or equal to all atoms in $N$:* $\forall A \in \mathsf{atoms}_{\mathsf{cls}}\ N.\ A \preceq_{\mathsf{t}} \beta$
   - *$S$ is reachable from the initial state:* $\langle N; \{\}; \{\}; \epsilon; \dagger\rangle \Rightarrow^*_{\mathsf{ORD\text{-}RES\text{-}11}} S$
   - *there is a transition from $S$ to $S'$:* $S \Rightarrow_{\mathsf{ORD\text{-}RES\text{-}11}} S'$

   *then there is a corresponding regular SCL(FOL) transition:* $(proj_{state}\ S) \Rightarrow^{proj_{clss}\ N,\ proj_{trm}\ \beta}_{Reg\text{-}SCL} (proj_{state}\ S')$

Instead of assuming that $\beta$ is greater than or equal to all atoms in $N$, we could have fixed it to be the greatest atom of $N$. We preferred the slightly more general formulation because it more closely follows SCL(FOL), which does not restrict $\beta$ to be an atom of $N$.

After having proven Theorem 7.37, we went back to the formalization of SCL(FOL) [40] and generalized the main nonredundancy and termination results to our new concept of projectable strategy. We then instantiated these generalized results and obtained the following corollaries for free.

**Corollary 7.38** (ORD-RES-11 Nonsubsumption)**.** *Let $N$ be a finite set of ground clauses. If a run of the ORD-RES-11 calculus starting from $\langle N; \{\}; \{\}; \epsilon; \dagger\rangle$ learns a clause, then this clause is not subsumed by any of the initial or learned clauses.*

**Corollary 7.39** (ORD-RES-11 Termination)**.** *Let $N$ be a finite set of ground clauses. Any run of the ORD-RES-11 calculus starting from $\langle N; \{\}; \{\}; \epsilon; \dagger\rangle$ terminates.*

Corollaries 7.38 and 7.39 also hold for ORD-RES-1 thanks to the bisimulation proven in Theorem 7.35.

## 7.8 Conclusion

We proved in Isabelle/HOL that SCL(FOL) can simulate ground ordered resolution and the other way around. We first formalized the ground ordered resolution calculus by adapting an existing formalization of ground superposition. We then reused the model-based strategy for ordered resolution specified by Bromberger et al. and performed ten refinement steps, proving a bisimulation between the steps. To ease these refinement steps, we proved a lifting lemma to get a bisimulation from a simulation. We then showed that the ultimate calculus of the last refinement step is a strategy of regular SCL(FOL). In summary, we have one strategy for ordered resolution, one strategy for regular SCL(FOL), and a bisimulation proof that the two strategies are equivalent.

The refinement steps we ended up with are not the only possible. They are the result of us trying to balance conflicting goals: we wanted to minimize the differences between each steps, which minimizes the complexity of the matching relations and the simulation proofs, but we also wanted to minimize the number of steps because each step adds some overhead (e.g., proving that the transition system is well-behaved).

The Isabelle/HOL formalization was invaluable because it forced us to consider corner cases (e.g., as described for ORD-RES-7) and allowed us to notice small mistakes in the initial specification of some of the calculi. The tooling for proof automation, such as sledgehammer and order, helped us tremendously by allowing us quickly discharge simpler subgoals and focus on the big picture.

# Part III

# Simulations between Virtual Machines

# Chapter 8

# Optimizing Virtual Machines

This chapter is based on a conference paper coauthored with Stefan Brunthaler [46]. It describes my formalization work.

## 8.1 Introduction

Every day, every person with a computer or smartphone executes, knowingly or not, an enormous amount of JavaScript. Confident that the machinery executing JavaScript works correctly, we use it day in and day out. A closer look at the correctness of JavaScript virtual machines shows that this confidence is unwarranted. Through abuse of implementation errors, attackers hijack victim devices through arbitrary code execution. In 2020, Google Project Zero raised public awareness for this danger in a three-part series of blog articles on so-called "JITSploitation" [58, 59, 60].

This should not come as a surprise, particularly as prior research by Yang et al. has already looked at the prevalence of implementation errors in compilers [127]. Their comparison of the LLVM, GCC, and CompCert compilers provides strong evidence of the effectiveness of formalization and verification to reduce implementation errors.

To establish confidence in the JavaScript computing machinery, one could replicate the CompCert [74] effort for a JavaScript virtual machine. Prior research by Myreen has shown that this approach is nontrivial [84]. Just-in-time compilers rely on self-modification and speculative optimizations to speed up programs. Both of these optimization techniques are at odds with the CompCert approach.

An alternative strategy to overcome these obstacles would be to sidestep just-in-time compilation and focus on interpreters instead. The expected advantages are ease of implementation, no self-modification, and no dynamic generation of machine-native code. Together, these advantages would also simplify the formalization and verification process.

But what kind of impact would such a strategy have on performance? Conventional wisdom states that interpreters are slow, and that performance requires just-in-time compilation. Prior research in interpreter optimization, however, reports remarkable and important speedups [31, 32, 51, 120, 126].

In this chapter, we build on prior results in interpreter optimization to formalize two speculative optimizations: inline caching and unboxing. Our formalization focuses on optimizing a stack-based interpreter for a small bytecode language support-

ing operand-stack manipulation, dynamic memory manipulation, arbitrary built-in operations, conditional jump, and (possibly recursive) function calls. The features were selected to be representative of the virtual machines of many other popular languages such as Lua, Perl, Python, and Ruby. The techniques used in the formalization could serve as a basis for the construction of efficient and correct virtual machine interpreters for such languages.

While such verifiably correct interpreters would not match the peak performance of their highly optimized just-in-time compiled counterparts, they should offer acceptable performance for a wide variety of tasks and computational needs. We believe, therefore, that using these efficient and verifiably correct interpreters would be preferable in safety-critical, high-assurance contexts. In addition, these interpreters could form the backbone of a secure, trusted infrastructure that we can rely upon when new just-in-time compiler bugs are exploited in the wild.

Summing up, this chapter makes the following contributions:

- We present a formalization of Dyn: a stack-based interpreter for a small dynamically typed bytecode language.
- We present a formalization of Inca: a self-optimizing extension of Dyn that uses advanced type feedback via inline caching.
- We present a formalization of Ubx: a self-optimizing extension of Inca that allows the manipulation of data in machine-native representation.
- Our formalization abstracts over several details (e.g., the built-in operations) and can be instantiated for several concrete languages.
- We prove both speculative optimizations to be sound through a bisimulation between Dyn and Inca and a bisimulation between Inca and Ubx.
- We show exemplary optimizing compilation passes, prove their soundness, and discuss their completeness.

Our formalization was developed using the Isabelle/HOL proof assistant. It consists of approximately 6500 nonblank lines[1] and is publicly available in the *Archive of Formal Proofs* [39]. As a sanity check, all locales defined in this formalization were instantiated with suitable examples to ensure that the assumptions are consistent.

Since completion of the project as described in this chapter, the formalization was expanded to support functions with multiple return values, function-local variables, and conditional jumps to labels instead of numeric positions. It was also expanded with a completeness proof of the compilation between Inca and Ubx w.r.t. some well-formedness criteria. The interested reader can refer to the formalization for more details.

## 8.2 Background

We briefly introduce virtual machines, the overhead of dynamic typing, and the overhead of boxed data objects.

**Virtual Machines** In the context of programming languages, a virtual machine is a piece of software used to execute a program without having to compile it to

[1] Counted using `grep -Ev '^[[:blank:]]*$'`.

machine code ahead of time. Virtual machines can thus abstract over the concrete hardware machines (e.g., RISC-V, ARM64, or AMD64) and offer a "write once, run anywhere" promise; though a more accurate description would be "write once, run anywhere a corresponding virtual machine is available". Examples of virtual machines include the CPython engine, the Java Virtual Machine, the Lua Virtual Machine, and the V8 engine (typically used for JavaScript or for WebAssembly). A virtual machine does not usually operate directly on the source code of the program (or an abstract syntax tree thereof) but rather on some intermediate representation called a bytecode language.

A bytecode language is an instruction set specifically designed to be efficiently executed by a virtual machine. Examples include the Java bytecode, the Python bytecode, and WebAssembly. A program written in a programming language must be first compiled to the appropriate bytecode language; this can be done either *ahead of time* (e.g., as done with Java programs), or as a preprocessing step of the virtual machine (e.g., as done by CPython). Because a bytecode language is usually higher-level than instruction sets for hardware machine, this compilation step can usually be done efficiently.

**Overhead of Dynamic Typing** Consider the following (slightly modified) implementation of the add operation in JavaScriptCore, the JavaScript implementation of the open-source WebKit browser engine used in Apple's Safari web browser.

```
1  JSValue jsAdd(JSGlobalObject* global, JSValue v1, JSValue v2) {
2    // Two numbers is the most common case
3    if (v1.isNumber() && v2.isNumber()) {
4      return jsNumber(v1.asNumber() + v2.asNumber());
5    }
6
7    // A string and a nonobject is also quite common
8    if (v1.isString() && !v2.isObject()) {
9      if (v2.isString()) {
10       return jsString(global, asString(v1), asString(v2));
11     }
12     String s2 = v2.toWTFString(global);
13     return jsString(global, asString(v1), s2);
14   }
15
16   // All other cases are pretty uncommon
17   return jsAddSlowCase(global, v1, v2);
18 }
```

The dynamically-typed add operation resolves concrete type assignments according to the expected frequency. First, JavaScriptCore delegates to C++'s addition operator when both operands, `v1` and `v2`, are numeric (lines 3–5). Second, JavaScriptCore performs string concatenation, including coercion of the second operand, when the first operand is a string (lines 8–14). Third, JavaScriptCore delegates implementation to `jsAddSlowCase` in all other cases (line 17), which is deemed "pretty uncommon" in the actual and original source code comment on line 16.

The cases differ in the number of type checks and branching operations. In the following, we assume that if-statements branch only when the condition is false. When both operands have a numeric type, two type checks are performed and no

branching is required. When the first operand is a string, there are five type checks (i.e., two checks for numbers, two checks for strings, and one check for nonobject) and one to two branches (i.e., one branch after the failed check for two numbers plus a second branch if the second argument is not a string and needs to be coerced). In all other cases, the operation execution requires four type checks (i.e., two checks for numbers, one check for string, and one check for object) and two branches (i.e., one branch after the failed check for two numbers plus a second branch after the failed check for a string and a nonobject) before delegating to the function `jsAddSlowCase`, which requires some more type checks and branches to determine less likely type assignments.

This implementation of the add operator incurs a performance penalty when type assignment expectations are not met. Consider a frequently executed, tight loop with a single string concatenation:

```
1 result = "This is a string";
2 for (i = 0; i < 100000; i += 1) {
3    result += i;
4 }
```

In this example, each iteration of the loop incurs five type checks, two branches, and a type coercion for the add operation to concatenate a string and a number. These type checks and branches are redundant because, for each loop iteration, the variable `result` always refers to a string and the variable `i` always refers to a number. If the case for string concatenation was ranked first in the implementation of `jsAdd`, then only three type checks, one branch, and one type coercion would be required. The downside would be that adding two numbers would suffer from two more type checks and one branch.

The effect of suboptimal static type-encoding in operation implementations of dynamic languages, as illustrated by the example above, has been known for decades. In 1982, Baden analyzed Smalltalk code and discovered what he termed a "dynamic locality of type usage." [7] In their landmark paper from 1984, Deutsch and Schiffman described what was to become one, if not the most, important optimization techniques to address this problem: *inline caching* [49]. In its original form, inline caching means that the virtual machine directly overwrites the target address of a call instruction in memory. So instead of calling the default routine that checks the types of all parameters (e.g., the type-generic `jsAdd` function) one would overwrite the address of the call instruction to type-dependent function, prefixed with so-called guards (i.e., type checks to ensure that the expected types were passed). As a result, a subsequent execution of the same instruction will "short-circuit" the type checks of the original routine and merely guard against expected types.

**Overhead of Boxed Data Objects** Boxed objects wrap primitive data types, such as numbers or characters. One one hand, primitive data usually can be manipulated using efficient machine-native operations and data representations. They require, however, dedicated type-dependent operations (e.g., multiplying two integers requires a different machine-native operation from the one to multiply two floating point numbers). On the other hand, boxed objects can be easily stored in the heap, and all other operations can refer to them in a uniform way using references or addresses. Boxed objects, furthermore, simplify the implementation of custom object and type

systems.

"Boxing" primitive data involves replacing the data item with a reference to an object representing the primitive data item. The resulting boxed object can, therefore, not be directly manipulated: assume that two 32-bit integers are in boxed object representation, then a simple machine-native integer addition would add their addresses instead of their numeric values. To manipulate boxed objects, their wrapped primitive data need to be "unboxed" first.

Boxing and unboxing requires a surplus of computation: to access the wrapped data, the computer must resolve the data references in the boxed objects. Additional operations, most often related to automatic memory management, must be taken into consideration as well. In Python, e.g., each push operation that puts data onto the operand stack needs to adjust the object's reference count. Machine-native data, on the other hand, need not be reference counted, as they exist on their own in binary representation and need no automatic memory management.

With unboxing, data locality is improved, as the indirection via the boxed object wrapper is eliminated. Automatic memory management operations are reduced, as these operations are only required to manage boxed objects. Overall memory consumption can be reduced, because fewer objects are required. Automatic memory management techniques can be adjusted to take this into account. This effect is most pronounced on immediate memory management techniques such as reference counting.

## 8.3 Overview of the Formalization

Our formalization has three parts, each concerned with a separate bytecode language.

Dyn (Section 8.4) a stack-based interpreter for a small dynamically typed bytecode language; it provides a baseline for optimizations.

The features supported by Dyn are intentionally kept minimal, but include the most representative features found in existing virtual machine interpreters for dynamic languages: operand stack manipulation, dynamic memory manipulation, built-in operations, conditional jumps, and (possibly recursive) function calls.

Inca (Section 8.5) extends Dyn with a speculative optimization known as inline caching. This type-based optimization is embedded directly in the semantics and thus performed automatically at run time. If the encountered types of an inlined operation match our speculation, the optimization is said to be a *hit*. Otherwise, the optimization is said to be a *miss* and must be rolled back. To ensure the soundness of this speculative optimization, we define a relation between unoptimized Dyn and optimized Inca programs, and prove that it is a bisimilarity, meaning that the compiled program has the same behaviour as the unoptimized one and vice versa. In addition, we provide a simple compilation scheme and prove its soundness and completeness.

Ubx (Section 8.6) extends Inca with operations to manipulate unboxed, machine-native data. This optimization is also type-based but proceeds in two stages. First, an optimization pass rewrites the program *ahead of time* by substituting some type-generic instructions with type-specific alternatives that directly manipulate unboxed data. Second, the semantics is extended to perform the minimum number of checks at *run time* to ensure that the type-specific, optimized instructions rewritten in the

first stage operate on the expected types and roll the optimization back if needed. Again, the soundness of this optimization is based on a bisimilarity relation, this time between unoptimized Inca programs and optimized Ubx programs. We also provide an exemplary compilation scheme based on a simple static analysis, and prove its soundness. We finish by discussing the incompleteness of this compilation scheme and some possible way forward.

We strove to keep the languages highly general by abstracting over a variety of implementation considerations. The most important abstraction is concerned with built-in operations. Instead of fixing a small set of these operations (such as arithmetic and Boolean operations) and optimizing them, we instead define an algebra of operations. For any operation of the algebra's carrier set, we can (i) determine the operation's arity,[2] and (ii) evaluate the operation on the given arguments. The semantics of all three languages, therefore, need only to ensure that operations receive the correct number of arguments, and manipulate their results accordingly. By construction, this technique ensures that our formalization supports all operations, and we can mostly avoid arguing "without loss of generality."

## 8.4 Dyn: Interpreter for Dynamically Typed Languages

The Dyn interpreter corresponds to a simple, stack-based bytecode interpreter to execute a dynamically typed programming language.

### 8.4.1 Syntax and Semantics

**Identifiers** The identifiers for variables and functions are members of the abstract types *'var* and *'fun*, respectively.

**Values** The manipulated values belong to the abstract type *'dyn* and are specified in the locale dynval. Locales are Isabelle's module system to define hierarchies of parametric theories [8]; they are described in Section 2.5.

**locale** dynval =
  **fixes**
    *is_true* :: *'dyn* ⇒ *bool* **and**
    *is_false* :: *'dyn* ⇒ *bool*
  **assumes**
    ∀*x*. ¬(*is_true* *x* ∧ *is_false* *x*)

In the locale dynval, *'dyn* is a fixed abstract type, *is_true* and *is_false* are locale parameters, and the last line states the locale assumption. The predicates *is_true* and *is_false* identify values that should be considered "true" and "false" respectively by Dyn's semantics (e.g., many programming languages consider the number zero to be "false" and nonzero numbers to be "true"). The assumption ensures that a value cannot be considered both "true" and "false". Note that a value could be neither "true" nor "false".

[2] All operations always return exactly one result. An operation with an arity of zero is equivalent to a constant.

**Operations** The built-in operations are represented by the type *'op* of the locale nary_operations.

**locale** nary_operations =
  **fixes**
    *eval* :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and**
    *arity* :: *'op* ⇒ ℕ

The function *arity* expresses the arity of an operation and the function *eval* evaluates an operation on an argument list. The length of the argument list must correspond to the arity of the operation; this is enforced by the semantics of our three interpreters.

**Environments** The locale env represents the generic notion of an environment as a partial mapping from keys to values.

**locale** env =
  **fixes**
    *empty* :: *'env* **and**
    *add* :: *'env* ⇒ *'key* ⇒ *'val* ⇒ *'env* **and**
    *get* :: *'env* ⇒ *'key* ⇒ *'val option* **and**
    *list* :: *'env* ⇒ (*'key* × *'val*) *list*
  **assumes**
    ∀*k*. *get empty k* = † **and**
    ∀*e k v*. *get* (*add e k v*) *k* = *v* **and**
    ∀*e k*$_1$ *k*$_2$ *v*. *k*$_1$ ≠ *k*$_2$ ⟶ *get* (*add e k*$_1$ *v*) *k*$_2$ = *get e k*$_2$ **and**
    ∀*e*. *get e* = map_of (*list e*) **and**
    ∀*e*. distinct (map fst (*list e*))

The environment is represented by the abstract type *'env*. The empty environment is represented by the constant *empty*. The function *add* extends an environment by associating a key with a value; this overrides any previous association of this key. The function *get* queries the value associated with a key; the result is † if no value was associated with this key. The function *list* maps an environment to a model represented as a list of key-value pairs; the idea is that *'env* should behave like its model. The three first assumptions express the semantics of the *empty* and *add* in terms of *get*: there is no value associated to any key in the empty environment, adding a key-value association makes it available for retrieval, and adding a key-value association only affects one key. The penultimate assumption expresses the semantics of *get* in terms of *list*: applying *get* to an environment must be equivalent to applying map_of to the environment's model as a list of key-value pairs. The Isabelle/HOL function map_of :: (*'a* × *'b*) *list* ⇒ *'a* ⇒ *'b option* traverses a list of key-value pairs and searches for the first value associated with a key. The ultimate assumption states that the model as list of key-value pairs must contain only distinct keys; this makes it easier to work with the map_of function.

In the following, we will use two instantiations of the env locale. One instantiation will store function definitions and instantiate the type *'env* with *'fenv*, the type *'key* with *'fun*, and the type *'val* with *fundef*; we will add the subscript *fun* to the names

| | | |
|---|---|---|
| *instr* | ::= Push *'dyn* \| Pop \| | *stack manipulation* |
| | Load *'var* \| Store *'var* \| | *memory manipulation* |
| | Op *'op* \| | *operations on data* |
| | CJump $\mathbb{N}$ \| | *conditional jump* |
| | Call *'fun* | *function call* |
| *fundef* | $=$ *instr list* $\times$ $\mathbb{N}$ | *function definition* |
| *prog* | $=$ *'fenv* $\times$ *'henv* $\times$ *'fun* | *program definition* |

Figure 8.1: The static representation of Dyn. The type *'fenv* is an environment that maps function symbols (type *'fun*) to function definitions (type *fundef*).

| | | |
|---|---|---|
| *frame* | $=$ *'fun* $\times$ $\mathbb{N}$ $\times$ *'dyn list* | *stack frame* |
| *state* | $=$ *'fenv* $\times$ *'henv* $\times$ *frame list* | *program state* |

Figure 8.2: The dynamic representation of Dyn. The type *'fenv* is an environment that maps function symbols (type *'fun*) to function definitions (type *fundef*).

of the locale parameters (e.g., the function $get_{fun}$ of this instantiation corresponds to the function *get* of the locale). The other instantiation will model dynamic memory and instantiate the type *'env* with *'henv*, the type *'key* with *'var* $\times$ *'dyn* and the type *'val* with *'dyn*; we will add the subscript *mem* to the names of the locale parameters (e.g., the function $get_{mem}$ of this instantiation corresponds to the function *get* of the locale).

**Static representation** Figure 8.1 shows the static representation of the Dyn language. *Instructions* belong to one of the following categories: manipulation of the operand stack, manipulation of the dynamic memory, built-in operations, conditional jump, and function call. A *function definition* contains a list of instructions and the function's arity; the former can be extracted with the function body :: *fundef* $\Rightarrow$ *instr list* and the latter can be extracted with the function $\mathsf{arity}_{\mathsf{fun}}$ :: *fundef* $\Rightarrow$ $\mathbb{N}$. A *program* contains an initial environment to store function definitions, an initial environment that models the dynamic memory, and the identifier of the function with which to start execution; this initial function must have an arity of zero.

**Dynamic states** Figure 8.2 shows the dynamic representation of the Dyn language. A stack *frame* contains the identifier of the function it refers to, a program counter relative to the beginning of the function, and an operand stack represented as a list of dynamic values. A dynamic program *state* contains an environment to store function definitions, an environment to model the dynamic memory, and a nonempty call stack represented as a list of stack frames.

**Loading and initial states** The binary relation $\mathsf{load}_{\mathrm{Dyn}}$ associates the static representation of a program to an initial dynamic state. More precisely, the relation initializes the program state, obtains the initializing function from the program, and transfers control to this function.

$$\forall F\ f\ fd\ H.\ get_{fun}\ F\ f = fd \longrightarrow \mathsf{arity}_{\mathsf{fun}}\ fd = 0 \longrightarrow \mathsf{load}_{\mathrm{Dyn}}\ \langle F; H; f\rangle\ \langle F; H; \langle f; 0; \epsilon\rangle\rangle$$

**Final states** The predicate $\mathsf{final}_{\mathrm{Dyn}}$ identifies final states as the ones having a call stack with a single stack frame, where the program counter points beyond the last instruction.

$$\begin{aligned}&\forall F\ H\ \Phi.\ \mathsf{final}_{\mathrm{Dyn}}\ \langle F; H; \Phi\rangle \longleftrightarrow\\ &\quad(\exists f\ pc\ \Sigma\ fd.\ \Phi = \langle f; pc; \Sigma\rangle \wedge get_{fun}\ F\ f = fd \wedge pc = |\mathsf{body}\ fd|)\end{aligned}$$

**Operational semantics** The operational semantics is defined by the small-step transition relation $\rightarrow_{\mathrm{Dyn}}$ between program states. The semantics of most instructions corresponds to the well-known, standard behaviour. The dynamic memory is partitioned by variable names, which are statically encoded in the load and store instructions, and each partition may contain any number of dynamic values, which are indexed by a dynamic value taken from the operand stack.

We first present the rules for stack manipulation:

**Push** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, d\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Push}\ d$

**Pop** $\langle F; H; \Phi, \langle f; pc; \Sigma, d\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Pop}$

We now present the rules for memory manipulation:

**Load** $\langle F; H; \Phi, \langle f; pc; \Sigma, d_1\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, d_2\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Load}\ x$
4. $get_{mem}\ H\ \langle x; d_1\rangle = d_2$

**Store** $\langle F; H; \Phi, \langle f; pc; \Sigma, d_2, d_1\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H'; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$

3. $(\mathsf{body}\ fd)[pc] = \mathsf{Store}\ x$
4. $H' = add_{mem}\ H\ \langle x; d_1\rangle\ d_2$

We now present the rules for operations on data:

**Op** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Op}\ op$
4. $arity\ op \leq |\Sigma|$
5. $\Sigma' = (\mathsf{drop}\ (arity\ op)\ \Sigma), (eval\ op\ (\mathsf{take}\ (arity\ op)\ \Sigma))$

The rule Op assumes with the side condition 4 that there are enough arguments on the operand stack before evaluating the operation. This assumption ensures that the list $\mathsf{take}\ (arity\ op)\ \Sigma$ is of length $arity\ op$, which is the arity of the operand $op$.

We now present the rules for conditional jump:

**CJump-True** $\langle F; H; \Phi, \langle f; pc; \Sigma, d\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; n; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{CJump}\ n$
4. $is\_true\ d$

**CJump-False** $\langle F; H; \Phi, \langle f; pc; \Sigma, d\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{CJump}\ n$
4. $is\_false\ d$

The rule CJump-True transfers the control flow to a position relative to the beginning of the function. Note that execution gets stuck if a jump condition represents neither true nor false.

The rule CJump-False only increments the program counter so that execution will continue at the next instruction.

We now present the rules for function call:

**Fun-Call** $\langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle\rangle \rightarrow_{\mathrm{Dyn}} \langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle, \langle g; 0; \Sigma_g\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc_f < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc_f] = \mathsf{Call}\ g$
4. $get_{fun}\ F\ g = gd$
5. $\mathsf{arity}_{\mathsf{fun}}\ gd \leq |\Sigma_f|$
6. $\Sigma_g = \mathsf{take}\ (\mathsf{arity}_{\mathsf{fun}}\ gd)\ \Sigma_f$

**Fun-End** $\langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle, \langle g; pc_g; \Sigma_g\rangle\rangle \rightarrow_{\text{Dyn}} \langle F; H; \Phi, \langle f; 1 + pc_f; \Sigma'_f\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ g = gd$
2. $pc_g = |\mathsf{body}\ gd|$
3. $|\Sigma_g| = 1$
4. $\mathsf{arity_{fun}}\ gd \leq |\Sigma_f|$
5. $\Sigma'_f = (\mathsf{drop}\ (\mathsf{arity_{fun}}\ gd)\ \Sigma_f), \Sigma_g$

The rule Fun-Call assumes with the side condition 5 that there are enough operands on the operand stack to call the function $g$. A new stack frame is created and the arguments copied to the operand stack of the new frame. Note that a function may call itself recursively.

The rule Fun-End proceeds in two steps. First, the remaining value on the called function's operand stack is interpreted as its result and its stack frame is discarded. Second, the arguments on top of the calling function's operand stack are replaced by the called function's result and the program counter is incremented.

**Example 8.1.** *Let main* :: *'fun be a function identifier. Let instrs* :: *instr list be a list of* Dyn *instructions. Let fahrenheit* :: *'var and celsius* :: *'var be two variable identifiers. Let sub* :: *'op, mul* :: *'op, and div* :: *'op be operations identifiers. Let F* :: *'fenv be an environment of function definitions. Let H* :: *'henv be an environment of dynamic memory (a.k.a. a heap). Consider instrs to be equal to the following list of instructions:*

Push 0
Load *fahrenheit*
Push 32
Op *sub*
Push 5
Op *mul*
Push 9
Op *div*
Push 0
Store *celsius*

*Consider* $F = \{main \mapsto \langle instrs; 0\rangle\}$. *Consider* $H = \{\langle fahrenheit; 0\rangle \mapsto 65.0\}$. *The* Dyn *program* $\langle F; H; main\rangle$ *is meant to convert a temperature of 65 degrees Fahrenheit to the corresponding temperature in Celsius (i.e., it computes* $celsius = (fahrenheit - 32) \times 5/9$*). It can be loaded to the dynamic state* $\langle F; H; \langle main; 0; \epsilon\rangle\rangle$.

| | | |
|---|---|---|
| *instr* | ::= $\cdots$ \| | *instructions from* Dyn |
| | OpInl *'opinl* | *inlined operations on data* |
| *fundef* | = *instr list* $\times \mathbb{N}$ | *function definition* |
| *prog* | = *'fenv* $\times$ *'henv* $\times$ *'fun* | *program definition* |

Figure 8.3: The static representation of Inca. The type *'fenv* is an environment that maps function symbols (type *'fun*) to function definitions (type *fundef*).

*Its execution goes as follows:*

$$
\begin{aligned}
& \langle F; H; \langle main; 0; \epsilon \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 1; 0 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 2; 65.0 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 3; 65.0, 32 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 4; 33.0 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 5; 33.0, 5 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 6; 165.0 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 7; 165.0, 9 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 8; 18.33 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; H; \langle main; 9; 18.33, 0 \rangle \rangle \\
\rightarrow_{\text{Dyn}} & \langle F; \{\langle fahrenheit; 0 \rangle \mapsto 65.0, \langle celsius; 0 \rangle \mapsto 18.33\}; \langle main; 10; \epsilon \rangle \rangle
\end{aligned}
$$

## 8.5 Inca: Inline Caching

The Inca interpreter extends Dyn with support for a single bytecode instruction for inline caching of operations.

### 8.5.1 Syntax and Semantics

The static representation of Inca is a proper superset of Dyn's syntax; the only addition is an instruction to inlined operations (Figure 8.3). The dynamic program representation does not change.

**Inlined operations** The built-in inlined operations are members of the type *'opinl* of the locale nary_operations_inl.

**locale** nary_operations_inl = nary_operations +
  **fixes**
    $eval_{inl}$ :: *'opinl* $\Rightarrow$ *'dyn list* $\Rightarrow$ *'dyn* **and**
    *inl* :: *'op* $\Rightarrow$ *'dyn list* $\Rightarrow$ *'opinl option* **and**
    *deinl* :: *'opinl* $\Rightarrow$ *'op* **and**
    $perf_{inl}$ :: *'opinl* $\Rightarrow$ *'dyn list* $\Rightarrow$ *bool*
  **assumes**

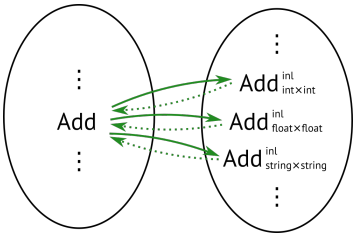



Figure 8.4: The relationship between elements of type *'op* (on the left) and elements of type *'opinl* (on the right). Solid arrows represent calls to *inl* and dotted arrows represent calls to *deinl*.

$$\forall op\ xs\ op_{inl}.\ inl\ op\ xs = op_{inl} \longrightarrow deinl\ op_{inl} = op \textbf{ and}$$
$$\forall op\ xs\ op_{inl}.\ inl\ op\ xs = op_{inl} \longrightarrow perf_{inl}\ op_{inl}\ xs \textbf{ and}$$
$$\forall xs\ op_{inl}.\ |xs| = arity\ (deinl\ op_{inl}) \longrightarrow eval_{inl}\ op_{inl}\ xs = eval\ (deinl\ op_{inl})\ xs$$

An operation of type *'op* may be mapped to any number (including none) of inlined operations of type *'opinl* with the function *inl*, which gives the most specific inlined operation for concrete operand types. This mapping can be inverted with the function *deinl*. Figure 8.4 illustrates the relationship between operations of type *'op* and operations of type *'opinl*.

The predicate $perf_{inl}$ informs us on whether evaluating an inlined operation on a operand list will be performant. It is related to the function *inl* by the second assumption, which states that the result of inlining an operation for an operand list must be performant. Why have the predicate $perf_{inl}$ and not always use the function *inl* to always have the optimal inlined operation? Because repeated calls to *inl* might be too costly. A typical implementation might start with a case analysis of the operation followed by a linear search for the most efficient inlined operation. This may be time consuming if the cardinality of *'op* or *'opinl* is high. We thus want to avoid calling *inl* if possible. The predicate $perf_{inl}$ has a simpler responsibility: to identify whether an inlined operation is performant enough. A typical implementation might start with a case analysis of the operation followed by a simple comparison of the operand types with the expected type—no need for a costly search as in the function *inl*. So when we already have an inlined operation and just want to ensure that it is still a good fit, we can use the efficient predicate $perf_{inl}$ instead of the costly function *inl*.

Finally, $eval_{inl}$ can be used to evaluate inlined operations on an operand list. Its result must be the same as evaluating *eval* on the corresponding deinlined operation when called with the correct number of operands. The semantics of Inca ensures that this is the case.

We define a helper function to express the arity of an inlined operation.

**Definition 8.2.** *The function* $\mathsf{arity}_{\mathsf{inl}} :: 'opinl \Rightarrow \mathbb{N}$ *expresses the arity of an inlined operation.*

$$\forall op_{inl}.\ \mathsf{arity}_{\mathsf{inl}}\ op_{inl} = arity\ (deinl\ op_{inl})$$

**Semantics** Inca's dynamic representation, its loading relation $\mathsf{load}_{\mathrm{Inca}}$, and its set of final states (identified by the predicate $\mathsf{final}_{\mathrm{Inca}}$) are all the same as their Dyn

counterparts. The transition relation $\rightarrow_{\text{Inca}}$ is very similar to $\rightarrow_{\text{Dyn}}$ except for the rule to evaluate an operation (i.e., rule Op) which differs from Dyn and for three new rules (i.e., Op-Inl, Op-Inl-Hit, Op-Inl-Miss) absent from Dyn.

We only present the rules for operations on data:

**Op** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Inca}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Op}\ op$
4. $arity\ op \leq |\Sigma|$
5. $inl\ op\ (\mathsf{take}\ (arity\ op)\ \Sigma) = \dagger$
6. $\Sigma' = (\mathsf{drop}\ (arity\ op)\ \Sigma), (eval\ op\ (\mathsf{take}\ (arity\ op)\ \Sigma))$

**Op-Inl** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Inca}} \langle F'; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Op}\ op$
4. $arity\ op \leq |\Sigma|$
5. $inl\ op\ (\mathsf{take}\ (arity\ op)\ \Sigma) = op_{inl}$
6. $\Sigma' = (\mathsf{drop}\ (arity\ op)\ \Sigma), (eval_{inl}\ op_{inl}\ (\mathsf{take}\ (arity\ op)\ \Sigma))$
7. $F' = add_{fun}\ F\ f\ (\mathsf{rewrite}\ fd\ pc\ (\mathsf{OpInl}\ op_{inl}))$

When executing an operation (i.e., instruction Op), *inl* is used to check if an inlined operation exists for the supplied arguments. If no inlined operation exists (rule Op), then the operation is evaluated with Op as was the case in Dyn. If an inlined operation exists (rule Op-Inl), then two things take place. First, we evaluate the operation with $eval_{inl}$. Second, we cache the search for an optimized inline operation by replacing the Op instruction with an optimized OpInl instruction in the function definition. As a result, any subsequent execution then "short-circuits" the check for an inlined operation.

**Op-Inl-Hit** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Inca}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{OpInl}\ op_{inl}$
4. $\mathsf{arity}_{\mathsf{inl}}\ op_{inl} \leq |\Sigma|$
5. $perf_{inl}\ op_{inl}\ (\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma)$
6. $\Sigma' = (\mathsf{drop}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma), (eval_{inl}\ op_{inl}\ (\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma))$

**Op-Inl-Miss** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Inca}} \langle F'; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$

2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{OpInl}\ op_{inl}$
4. $\mathsf{arity}_{\mathsf{inl}}\ op_{inl} \leq |\Sigma|$
5. $\neg perf_{inl}\ op_{inl}\ (\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma)$
6. $\Sigma' = (\mathsf{drop}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma), (eval_{inl}\ op_{inl}\ (\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma))$
7. $F' = add_{fun}\ F\ f\ (\mathsf{rewrite}\ fd\ pc\ (\mathsf{Op}\ (deinl\ op_{inl})))$

When executing an inlined operation (i.e., instruction OpInl), the efficient predicate $perf_{inl}$ is used to test whether or not it is still appropriate for the supplied arguments—it was presumably appropriate when it was originally inlined but the operands could have been of different types. If this is the case, then execution continues as expected using an optimized function (rule Op-Inl-Hit). Otherwise, we undo the optimization by replacing the optimized instruction with the generic, unoptimized instruction in the function definition (rule Op-Inl-Miss); here we could use either the evaluation function *eval* or $eval_{inl}$, because they are semantically equivalent and it is unknown which one would be more efficient.

### 8.5.2 Bisimulation between Dyn and Inca

When two programs of Dyn and Inca simulate each other, they only differ in the function definitions stored in their respective function environments. The dynamic memories, the call stacks, and the domains of the function environments are identical. Given two corresponding function definitions from Dyn and Inca, they may only differ by the potential use of inline operations.

We define the matching relation $\sim$, which inspects all corresponding instructions of function definitions and checks with the function *deinl* that an inlined operation maps to its corresponding regular operation.

We first prove a forward simulation between Dyn and Inca.

**Lemma 8.3** (Forward Simulation)**.** *Let $s_1$ and $s_1'$ be dynamic states of* Dyn*. Let $s_2$ be a dynamic state of* Inca*. If $s_1 \rightarrow_{\text{Dyn}} s_1'$ and $s_1 \sim s_2$, then there exists a dynamic state $s_2'$ such that $s_2 \rightarrow_{\text{Inca}} s_2'$ and $s_1' \sim s_2'$.*

**Lemma 8.4** (Forward Matching Final States)**.** *Let $s_1$ and $s_2$ be dynamic states of* Dyn *and* Inca *respectively. If $s_1 \sim s_2$ and $\mathsf{final}_{\text{Dyn}}\ s_1$, then $\mathsf{final}_{\text{Inca}}\ s_2$.*

We then prove a backward simulation between Dyn and Inca.

**Lemma 8.5** (Backward Simulation)**.** *Let $s_1$ be a dynamic state of* Dyn*. Let $s_2$ and $s_2'$ be dynamic states of* Inca*. If $s_2 \rightarrow_{\text{Inca}} s_2'$ and $s_1 \sim s_2$, then there exists a dynamic state $s_1'$ such that $s_1 \rightarrow_{\text{Dyn}} s_1'$ and $s_1' \sim s_2'$.*

**Lemma 8.6** (Backward Matching Final States)**.** *Let $s_1$ and $s_2$ be dynamic states of* Dyn *and* Inca *respectively. If $s_1 \sim s_2$ and $\mathsf{final}_{\text{Inca}}\ s_2$, then $\mathsf{final}_{\text{Dyn}}\ s_1$.*

Finally, we prove a bisimulation between Dyn and Inca.

**Theorem 8.7** (Bisimulation)**.** *There is a bisimulation between* Dyn *and* Inca *w.r.t. $\sim$.*

*Proof Sketch.* By Lemmas 8.3 to 8.6. □

Theorem 8.7 corresponds to an interpretation of the locale bisimulation from Chapter 6.

### 8.5.3 Compilation from Dyn to Inca

We define the function compile that compiles a static program representation of Dyn to a static program representation of Inca; it maps all Dyn-instructions found in the function definitions to their equivalent Inca-instructions.

We prove that compiled programs simulate their uncompiled counterparts.

**Lemma 8.8** (Compiled Matching States)**.** *Let $p_1$ and $p_2$ be static program representations of* Dyn *and* Inca *respectively. Let $s_1$ be a dynamic state of* Dyn*. If* $\mathsf{compile}\, p_1 = p_2$ *and* $\mathsf{load}_{\mathrm{Dyn}}\, p_1\, s_1$*, then there exists a state $s_2$ such that* $\mathsf{load}_{\mathrm{Inca}}\, p_2\, s_2$ *and $s_1 \sim s_2$.*

**Theorem 8.9** (Compilation Function)**.** *The function* compile *is a compiler from* Dyn *to* Inca*, which guarantees that the* Inca *program simulates the* Dyn *program.*

*Proof Sketch.* By Lemmas 8.5, 8.6 and 8.8. □

Theorem 8.9 corresponds to an interpretation of the locale compiler from Chapter 6. The simulation framework gives us a proof that the successful execution of a compiled Inca program has a behaviour equivalent to a behaviour of the original Dyn program. Remember from Chapter 6 that the infix relation $\Downarrow$ pairs a program to its run-time behaviour and that the ternary predicate rel_behaviour relates two behaviours w.r.t. a matching relation on dynamic states.

**Corollary 8.10** (Soundness of Compilation)**.** *Let $p_1$ and $p_2$ be static program representations of* Dyn *and* Inca *respectively. If $p_1$ compiles to $p_2$ (i.e.,* $\mathsf{compile}\, p_1 = p_2$*), and $p_2$ has behaviour $b_2$ (i.e., $p_2 \Downarrow b_2$), and $b_2$ is not a wrong behaviour, then $p_1$ has a behaviour that matches with $b_2$ (i.e.,* $\exists b_1.\ p_1 \Downarrow b_1 \wedge \mathsf{rel\_behaviour}\, (\sim)\, b_1\, b_2$*).*

Furthermore, we prove that compilation is complete for all loadable Dyn programs.

**Theorem 8.11** (Completeness of Compilation)**.** *Let $p_1$ be a static program representation of* Dyn *and $s_1$ a dynamic state of* Dyn*. If $p_1$ can be loaded into $s_1$ (i.e.,* $\mathsf{load}_{\mathrm{Dyn}}\, p_1\, s_1$*), then $p_1$ can be compiled to an* Inca*-program that can be loaded to an* Inca*-state matching $s_1$ (i.e.,* $\exists p_2\, s_2.\ \mathsf{compile}\, p_1 = p_2 \wedge \mathsf{load}_{\mathrm{Inca}}\, p_2\, s_2 \wedge s_1 \sim s_2$*).*

## 8.6 Ubx: Operations on Unboxed Data

The Ubx interpreter extends Inca with the concept of manipulating unboxed data representations.

### 8.6.1 Syntax and Semantics

The syntax of Ubx is a proper superset of Inca's syntax (Figure 8.6). The static representation of Ubx is a proper superset of Inca's syntax (Figure 8.5); the only additions are instructions to manipulate unboxed values. The dynamic program representation has an operand stack of unboxed values (Figure 8.6).

| | | |
|---|---|---|
| *type* | ::= $\mathsf{U}_1 \mid \mathsf{U}_2$ | *unboxed types* |
| *instr* | ::= $\cdots \mid$ | *instructions from* Inca |
| | $\mathsf{PushUbx}_1\ 'ubx_1 \mid$ | *stack manipulation* |
| | $\mathsf{PushUbx}_2\ 'ubx_2 \mid$ | *of unboxed data* |
| | $\mathsf{LoadUbx}\ type\ 'var \mid$ | *memory manipulation* |
| | $\mathsf{StoreUbx}\ type\ 'var \mid$ | *of unboxed data* |
| | $\mathsf{OpUbx}\ 'opubx$ | *unboxed operations* |
| *fundef* | $= instr\ list \times \mathbb{N}$ | *function definition* |
| *prog* | $= 'fenv \times 'henv \times 'fun$ | *program definition* |

Figure 8.5: The static representation of Ubx. The type *'fenv* is an environment that maps function symbols (type *'fun*) to function definitions (type *fundef*).

| | | |
|---|---|---|
| *ubx* | ::= $\mathsf{Dyn}\ 'dyn \mid$ | *boxed dynamic value* |
| | $\mathsf{Ubx}_1\ 'ubx_1 \mid$ | *unboxed value 1* |
| | $\mathsf{Ubx}_2\ 'ubx_2 \mid$ | *unboxed value 2* |
| *frame* | $= 'fun \times \mathbb{N} \times ubx\ list$ | *stack frame* |
| *state* | $= 'fenv \times 'henv \times frame\ list$ | *program state* |

Figure 8.6: The dynamic representation of Ubx. The type *'fenv* is an environment that maps function symbols (type *'fun*) to function definitions (type *fundef*).

**Values** The values manipulated through the operand stack may either be boxed or unboxed. In principle, any fixed number of unboxed types may be supported but, due to Isabelle/HOL not supporting abstractions over arbitrary numbers of types, we consider without loss of generality two unboxed types $'ubx_1$ and $'ubx_2$ (e.g., they could represent the 64-bit integers and 64-bit floating point numbers native to the hardware).

Because the operand stack may only contain values of a uniform type, we define the tagged union type *ubx* with three constructors: Dyn represents a boxed value while both $\mathsf{Ubx_1}$ and $\mathsf{Ubx_2}$ represent unboxed values. We extract a value stored in a value of type *ubx* by casting it to the desired type. Casting (i) checks that the *ubx* value is tagged with the expected constructor for the given type, and (ii) returns the unboxed value.

**Definition 8.12.** *The function* $\mathsf{cast_{Dyn}}$::*ubx* ⇒ $'dyn$ *option extracts the dynamic value out of its argument of type ubx; it fails and evaluates to* † *if its argument does not contain a dynamic value.*

$$\begin{aligned} \forall d.\ \mathsf{cast_{Dyn}}\ (\mathsf{Dyn}\ d) &= d \\ \mathsf{cast_{Dyn}}\ \mathsf{Ubx_1} &= \dagger \\ \mathsf{cast_{Dyn}}\ \mathsf{Ubx_2} &= \dagger \end{aligned}$$

The functions $\mathsf{cast_{ubx_1}}$::*ubx* ⇒ $'ubx_1$ *option* and $\mathsf{cast_{ubx_2}}$::*ubx* ⇒ $'ubx_2$ *option* are analogous.

Our formalization proves that casts are always successful and a real implementation of this optimization could thus omit these checks.

The boxing and unboxing operations are abstracted over in the locale unboxedval.

**locale** unboxedval = dynval +
  **fixes**
    $box_1$ :: $'ubx_1$ ⇒ $'dyn$ **and**
    $unbox_1$ :: $'dyn$ ⇒ $'ubx_1$ *option* **and**
    $box_2$ :: $'ubx_2$ ⇒ $'dyn$ **and**
    $unbox_2$ :: $'dyn$ ⇒ $'ubx_2$ *option*
  **assumes**
    $\forall d\ u_1.\ unbox_1\ d = u_1 \longrightarrow box_1\ u_1 = d$ **and**
    $\forall d\ u_2.\ unbox_2\ d = u_2 \longrightarrow box_2\ u_2 = d$

The function $box_1$ converts a machine-native value of type $'ubx_1$ to a dynamic value of type $'dyn$. The function $unbox_1$ undoes boxing by converting a dynamic value of type $'dyn$ to a machine-native value of type $'ubx_1$; this might fail by evaluating to † if the provided dynamic value is not of the expected type. The functions $box_2$ and $unbox_2$ are analogous but operate on machine-native values of type $'ubx_2$. The locale assumptions express that unboxing a dynamic value can be reverted and the resulting dynamic value is the same as the original.

We define a few functions to uniformly manipulate values of type *ubx* without having to separately consider cases for $'ubx_1$ and $'ubx_2$. We first define the type *type* which has one constant per unboxed type: The constant $\mathsf{U_1}$ is associated with the type $'ubx_1$ and the constant $\mathsf{U_2}$ is associated with the type $'ubx_2$. The coming

definitions use the function map_option :: $('a \Rightarrow 'b) \Rightarrow 'a\ option \Rightarrow 'b\ option$ from the Isabelle/HOL distribution, which applies a provided function to an optional value.

**Definition 8.13.** *The function* cast_box :: *type* $\Rightarrow$ *ubx* $\Rightarrow$ $'dyn$ *option casts an unboxed value to the requested type and then boxes the result to a dynamic value.*

$$\forall u.\ \mathsf{cast\_box}\ \mathsf{U}_1\ u = \mathsf{map\_option}\ box_1\ (\mathsf{cast}_{\mathsf{ubx}_1}\ u)$$
$$\forall u.\ \mathsf{cast\_box}\ \mathsf{U}_2\ u = \mathsf{map\_option}\ box_2\ (\mathsf{cast}_{\mathsf{ubx}_2}\ u)$$

**Definition 8.14.** *The function* unbox :: *type* $\Rightarrow$ $'dyn$ $\Rightarrow$ *ubx option unboxes a dynamic value to the requested type and then stores the result in an element of type ubx.*

$$\forall d.\ \mathsf{unbox}\ \mathsf{U}_1\ d = \mathsf{map\_option}\ \mathsf{Ubx}_1\ (unbox_1\ d)$$
$$\forall d.\ \mathsf{unbox}\ \mathsf{U}_2\ d = \mathsf{map\_option}\ \mathsf{Ubx}_2\ (unbox_2\ d)$$

**Definition 8.15.** *The function* norm :: *ubx* $\Rightarrow$ $'dyn$ *boxes an unboxed value to a dynamic value.*

$$\forall d.\ \mathsf{norm}\ (\mathsf{Dyn}\ d) = d$$
$$\forall u_1.\ \mathsf{norm}\ (\mathsf{Ubx}_1\ u_1) = box_1\ u_1$$
$$\forall u_2.\ \mathsf{norm}\ (\mathsf{Ubx}_2\ u_2) = box_2\ u_2$$

**Instructions** One new instruction per unboxed type pushes an unboxed constant onto the operand stack. Two generic instructions allow loading unboxed values from memory and storing them in memory respectively. Finally, one instruction manipulates unboxed, machine-native data. The number of new instructions to support $n$ unboxed types is thus $n + 3$.

**Operations on unboxed data** The built-in operations on unboxed data are members of the type $'opubx$ of the locale nary_operations_ubx.

**locale** nary_operations_ubx = nary_operations_inl + unboxedval +
**fixes**
$eval_{ubx}$ :: $'opubx \Rightarrow ubx\ list \Rightarrow ubx\ option$ **and**
$ubx$ :: $'opinl \Rightarrow type\ option\ list \Rightarrow 'opubx\ option$ **and**
$deubx$ :: $'opubx \Rightarrow 'opinl$ **and**
$typeof_{ubx}$ :: $'opubx \Rightarrow type\ option\ list \times type\ option$
**assumes**
$\forall op_{inl}\ types\ op_{ubx}.\ ubx\ op_{inl}\ types = op_{ubx} \longrightarrow deubx\ op_{ubx} = op_{inl}$ **and**
$\forall op_{ubx}\ xs\ y.\ eval_{ubx}\ op_{ubx}\ xs = y \longrightarrow$
$eval_{inl}\ (deubx\ op_{ubx})\ (\mathsf{map\ norm}\ xs) = \mathsf{norm}\ y$ **and**
$\forall op_{ubx}\ xs\ y.\ eval_{ubx}\ op_{ubx}\ xs = y \longrightarrow$
$inl\ (deinl\ (deubx\ op_{ubx}))\ (\mathsf{map\ norm}\ xs) = deubx\ op_{ubx}$ **and**
$\forall op_{ubx}.\ \mathsf{arity}_{\mathsf{inl}}\ (deubx\ op_{ubx}) = |\mathsf{fst}\ (typeof_{ubx}\ op_{ubx})|$ **and**
$\forall op_{inl}\ types\ op_{ubx}.\ ubx\ op_{inl}\ types = op_{ubx} \longrightarrow$
$\mathsf{fst}\ (typeof_{ubx}\ op_{ubx}) = types$ **and**
$\forall op_{ubx}\ xs\ \tau.\ typeof_{ubx}\ op_{ubx} = \langle \mathsf{map\ typeof}\ xs; \tau \rangle \longrightarrow$
$(\exists y.\ eval_{ubx}\ op_{ubx}\ xs = y \wedge \mathsf{typeof}\ y = \tau)$ **and**
$\forall op_{ubx}\ xs\ y.\ eval_{ubx}\ op_{ubx}\ xs = y \longrightarrow$
$typeof_{ubx}\ op_{ubx} = \langle \mathsf{map\ typeof}\ xs; \mathsf{typeof}\ y \rangle$

The function $eval_{ubx}$ evaluates an unboxed operation (of type $'opubx$) on unboxed arguments (of type $ubx$) using some efficient machine-native instructions. In contrast to $eval$ and $eval_{inl}$ which always succeed to calculate a result when given the correct number of arguments, $eval_{ubx}$ may fail and evaluate to † when applied to unboxed values of the wrong type.

The function $ubx$ maps an inlined operation (of type $'opinl$) on dynamic values to an unboxed operation (of type $'opubx$) on unboxed values. Instead of relying on the dynamic typing information extracted from the arguments at run time, $ubx$ relies on type information available at compilation time. For each argument, the provided type information takes the form of an optional unboxed type: $\mathsf{Ubx}_1$ means that an argument of type $'ubx_1$ is expected, $\mathsf{Ubx}_2$ means that an argument of type $'ubx_2$ is expected, and † means that an argument of type $'dyn$ is expected. The function $deubx$ inverses the mapping of $ubx$ and converts and unboxed operation back to an inlined operation.

The function $typeof_{ubx}$ evaluates to the type of a given unboxed operation: the result is a pair where the first element is the domain and the second element is the codomain.

We define a helper function to express the arity of an unboxed operation.

**Definition 8.16.** *The function* $\mathsf{arity}_{\mathsf{ubx}} :: 'opubx \Rightarrow \mathbb{N}$ *expresses the arity of an unboxed operation.*

$$\forall op_{ubx}.\ \mathsf{arity}_{\mathsf{ubx}}\ op_{ubx} = \mathsf{arity}_{\mathsf{inl}}\ (deubx\ op_{ubx})$$

**Semantics** The transition relation $\rightarrow_{\text{Ubx}}$ is similar to $\rightarrow_{\text{Inca}}$ except that the operand stack contains values of type $ubx$ and that it has supplementary rules to handle unboxed data. We show all rules here.

We first present the rules for stack manipulation:

**Push** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, \mathsf{Dyn}\ d\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Push}\ d$

**Push-Ubx$_1$** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, \mathsf{Ubx}_1\ u\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{PushUbx}_1\ u$

**Push-Ubx$_2$** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, \mathsf{Ubx}_2\ u\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{PushUbx}_2\ u$

**Pop** $\langle F; H; \Phi, \langle f; pc; \Sigma, u\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Pop}$

The rule Push wraps a dynamic value and pushes the result onto the stack. The rules Push-Ubx$_1$ and Push-Ubx$_2$ both wrap an unboxed value and push the result onto the stack. The rule Pop does not need any change; it is independent of the operand type.

We now present the rules for memory manipulation:

**Load** $\langle F; H; \Phi, \langle f; pc; \Sigma, u\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, \mathsf{Dyn}\ d_2\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Load}\ x$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u = d_1$
5. $get_{mem}\ H\ \langle x; d_1\rangle = d_2$

**Load-Ubx-Hit** $\langle F; H; \Phi, \langle f; pc; \Sigma, u_1\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma, u_2\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{LoadUbx}\ \tau\ x$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u_1 = d_1$
5. $get_{mem}\ H\ \langle x; d_1\rangle = d_2$
6. $\mathsf{unbox}\ \tau\ u_2 = u_2$

**Load-Ubx-Miss** $\langle F; H; \Phi, \langle f; pc; \Sigma, u_1\rangle\rangle \rightarrow_{\text{Ubx}} \langle F'; H; \Phi'\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{LoadUbx}\ \tau\ x$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u_1 = d_1$
5. $get_{mem}\ H\ \langle x; d_1\rangle = d_2$
6. $\mathsf{unbox}\ \tau\ u_2 = \dagger$
7. $F' = add_{mem}\ F\ f\ (\mathsf{generalize}\ fd)$
8. $\Phi' = \mathsf{box\_stack}\ f\ (\Phi, \langle f; 1 + pc; \Sigma, \mathsf{Dyn}\ d_2\rangle)$

**Store** $\langle F; H; \Phi, \langle f; pc; \Sigma, u_2, u_1\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H'; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$

3. $(\mathsf{body}\ fd)[pc] = \mathsf{Store}\ x$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u_1 = d_1$
5. $\mathsf{cast}_{\mathsf{Dyn}}\ u_2 = d_2$
6. $H' = add_{mem}\ H\ \langle x; d_1 \rangle d_2$

**Store-Ubx** $\langle F; H; \Phi, \langle f; pc; \Sigma, u_2, u_1 \rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H'; \Phi, \langle f; 1 + pc; \Sigma \rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{StoreUbx}\ \tau\ x$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u_1 = d_1$
5. $\mathsf{cast\_box}\ \tau\ u_2 = d_2$
6. $H' = add_{mem}\ H\ \langle x; d_1 \rangle d_2$

The rules for loading values from the dynamic memory distinguish three cases:

- a dynamic value is loaded and pushed directly on the operand stack;
- a dynamic value is loaded, successfully unboxed, and pushed on the operand stack; and
- a dynamic value is loaded, the unboxing fails, and the function is generalized to cancel the Ubx optimization.

All three rules first pop a value from the operand stack, and then cast it to a dynamic value, which is then used to index the dynamic memory.

In rule Load-Ubx-Miss, the unboxing fails because the dynamic value loaded from memory has a different type than what was expected when optimizing the program. Subsequent instructions expecting data in their machine-native representation cannot execute sensibly and must be generalized to cope with dynamic values. This generalization process applies to both the function definition and the call stack.

First, the function generalize generalizes the function definition by mapping all Ubx instructions to their Inca counterparts (e.g., it maps $\mathsf{PushUbx}_1$ to Push). For OpInl instructions, the function *deubx* identifies the corresponding *'opinl* operation.

Second, we need to update the operand stack to ensure that all elements use the boxed representation. If a tagged union contains an operand in unboxed data representation, these operands would not be accepted by the newly generalized instructions. To address this, we use the type information stored in the tagged union to box the object and replace the element with another tagged union (constructor Dyn) representing this newly boxed object. The operand stack of the current stack frame must be boxed, but so do the operand stacks of all other active stack frames of the same function. Because each stack frame only stores the identifier of the function, and each execution step retrieves the instruction from the function definition, all active function invocations will start to use the generalized instructions. The function box_stack does this by recursively traversing the call stack and generalizing the operand stack of all stack frames for function $f$; all other stack frames are left untouched.

The rules for storing values in memory all cast the operand on the top of the stack to the expected type and box it before storing it in memory. No rules are

needed to handle the case that an unboxed type does not match its expected type. The bisimulation relation proves that such a situation can never occur.

We now present the rules for operations on data:

**Op** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Op}\ op$
4. $arity\ op \leq |\Sigma|$
5. $\mathsf{traverse}\ \mathsf{cast}_{\mathsf{Dyn}}(\mathsf{take}\ (arity\ op)\ \Sigma) = ds$
6. $inl\ op\ ds = \dagger$
7. $\Sigma' = (\mathsf{drop}\ (arity\ op)\ \Sigma), (\mathsf{Dyn}\ (eval\ op\ ds))$

**Op-Inl** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F'; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{Op}\ op$
4. $arity\ op \leq |\Sigma|$
5. $\mathsf{traverse}\ \mathsf{cast}_{\mathsf{Dyn}}(\mathsf{take}\ (arity\ op)\ \Sigma) = ds$
6. $inl\ op\ ds = op_{inl}$
7. $\Sigma' = (\mathsf{drop}\ (arity\ op)\ \Sigma), (\mathsf{Dyn}\ (eval_{inl}\ op_{inl}\ ds))$
8. $F' = add_{fun}\ F\ f\ (\mathsf{rewrite}\ fd\ pc\ (\mathsf{OpInl}\ op_{inl}))$

**Op-Inl-Hit** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{OpInl}\ op_{inl}$
4. $\mathsf{arity}_{\mathsf{inl}}\ op_{inl} \leq |\Sigma|$
5. $\mathsf{traverse}\ \mathsf{cast}_{\mathsf{Dyn}}(\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma) = ds$
6. $perf_{inl}\ op_{inl}\ ds$
7. $\Sigma' = (\mathsf{drop}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma), (\mathsf{Dyn}\ (eval_{inl}\ op_{inl}\ ds))$

**Op-Inl-Miss** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\text{Ubx}} \langle F'; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{OpInl}\ op_{inl}$
4. $\mathsf{arity}_{\mathsf{inl}}\ op_{inl} \leq |\Sigma|$
5. $\mathsf{traverse}\ \mathsf{cast}_{\mathsf{Dyn}}(\mathsf{take}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma) = ds$

6. $\neg perf_{inl}\ op_{inl}\ ds$
7. $\Sigma' = (\mathsf{drop}\ (\mathsf{arity}_{\mathsf{inl}}\ op_{inl})\ \Sigma), (\mathsf{Dyn}\ (eval_{inl}\ op_{inl}\ ds))$
8. $F' = add_{fun}\ F\ f\ (\mathsf{rewrite}\ fd\ pc\ (\mathsf{Op}\ (deinl\ op_{inl})))$

**Op-Ubx** $\langle F; H; \Phi, \langle f; pc; \Sigma\rangle\rangle \rightarrow_{\mathrm{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma'\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{OpUbx}\ op_{ubx}$
4. $\mathsf{arity}_{\mathsf{ubx}}\ op_{ubx} \leq |\Sigma|$
5. $eval_{ubx}\ op_{ubx}\ (\mathsf{take}\ (\mathsf{arity}_{\mathsf{ubx}}\ op_{ubx})\ \Sigma) = u$
6. $\Sigma' = (\mathsf{drop}\ (\mathsf{arity}_{\mathsf{ubx}}\ op_{ubx})\ \Sigma), u$

The rules for evaluating regular and inlined operations require minimal adaptation: they must first cast their operands to dynamic values before evaluation. Again, no rule is required to handle an invalid cast, as our proof shows that such situations can never occur.

The new rule Op-Ubx does not need to perform any cast as it operates directly on unboxed data.

We now present the rules for conditional jump:

**CJump-True** $\langle F; H; \Phi, \langle f; pc; \Sigma, u\rangle\rangle \rightarrow_{\mathrm{Ubx}} \langle F; H; \Phi, \langle f; n; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{CJump}\ n$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u = d$
5. $is\_true\ d$

**CJump-False** $\langle F; H; \Phi, \langle f; pc; \Sigma, u\rangle\rangle \rightarrow_{\mathrm{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc; \Sigma\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc] = \mathsf{CJump}\ n$
4. $\mathsf{cast}_{\mathsf{Dyn}}\ u = d$
5. $is\_false\ d$

The rules CJump-True and CJump-False require minimal adaptation; they now cast their operand to a dynamic Boolean value.

We now present the rules for function call:

**Fun-Call** $\langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle\rangle \rightarrow_{\mathrm{Ubx}} \langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle, \langle g; 0; \Sigma_g\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ f = fd$
2. $pc_f < |\mathsf{body}\ fd|$
3. $(\mathsf{body}\ fd)[pc_f] = \mathsf{Call}\ g$

4. $get_{fun}\ F\ g = gd$
5. $\mathsf{arity_{fun}}\ gd \leq |\Sigma_f|$
6. $\forall u \in \mathsf{set}\ (\mathsf{take}\ (\mathsf{arity_{fun}}\ gd)\ \Sigma_f).\ \mathsf{typeof}\ u = \dagger$
7. $\Sigma_g = \mathsf{take}\ (\mathsf{arity_{fun}}\ gd)\ \Sigma_f$

**Fun-End** $\langle F; H; \Phi, \langle f; pc_f; \Sigma_f\rangle, \langle g; pc_g; \Sigma_g\rangle\rangle \rightarrow_{\text{Ubx}} \langle F; H; \Phi, \langle f; 1 + pc_f; \Sigma'_f\rangle\rangle$
Side conditions:

1. $get_{fun}\ F\ g = gd$
2. $pc_g = |\mathsf{body}\ gd|$
3. $|\Sigma_g| = 1$
4. $\forall u \in \mathsf{set}\ \Sigma_g.\ \mathsf{typeof}\ u = \dagger$
5. $\mathsf{arity_{fun}}\ gd \leq |\Sigma_f|$
6. $\Sigma'_f = (\mathsf{drop}\ (\mathsf{arity_{fun}}\ gd)\ \Sigma_f), \Sigma_g$

The rules Fun-Call and Fun-End require minimal adaptation; they now check that the function arguments or function return value are of the dynamic type before calling or ending the function.

### 8.6.2 Bisimulation between Inca and Ubx

The validity of a sequence of Ubx instructions can be statically verified by an abstract interpretation that calculates a form of strongest postcondition (i.e., the arity and types of values on the operand stack following the execution of the sequence). This means that, if a function is given the right number of boxed arguments, then it will successfully execute and return a value of the computed types. The strongest postcondition of an instruction takes a stack of types as input and calculates the stack of types resulting from executing that instruction.

Two corresponding program states from Inca and Ubx simulate each other (expressed by the $\approx$ binary relation) if they have the same dynamic memory, if both their function environments and call stacks are similar, and if an abstract interpretation of all function definitions succeeds.

Two function environments are similar if they have the same domain and if, given a function definition in Ubx and in Inca, the Ubx function definition generalizes to the Inca function definition.

Call stacks are similar if

- they have the same height,
- two corresponding stack frames refer to the same function, have the same program counters, and Ubx's operand stack may be boxed to Inca's,
- the abstract interpretation of the function up to the current program counter matches with the operand types in the stack, and
- the current instruction of all nontopmost stack frames must be a call instruction to the function of the stack frame immediately above.

We first prove a forward simulation between Inca and Ubx.

**Lemma 8.17** (Forward Simulation)**.** *Let $s_1$ and $s_1'$ be dynamic states of* Inca*. Let $s_2$ be a dynamic state of* Ubx*. If $s_1 \approx s_2$ and $s_1 \rightarrow_{\text{Inca}} s_1'$, then there exists a dynamic state $s_2'$ such that $s_2 \rightarrow_{\text{Ubx}} s_2'$ and $s_1' \approx s_2'$.*

**Lemma 8.18** (Forward Matching Final States)**.** *Let $s_1$ and $s_2$ be dynamic states of* Inca *and* Ubx *respectively. If $s_1 \approx s_2$ and* $\mathsf{final}_{\text{Inca}}\ s_1$*, then* $\mathsf{final}_{\text{Ubx}}\ s_2$*.*

We then prove a backward simulation between Inca and Ubx.

**Lemma 8.19** (Backward Simulation)**.** *Let $s_1$ be a dynamic state of* Inca*. Let $s_2$ and $s_2'$ be dynamic states of* Ubx*. If $s_1 \approx s_2$ and $s_2 \rightarrow_{\text{Ubx}} s_2'$, then there exists a dynamic state $s_1'$ such that $s_1 \rightarrow_{\text{Inca}} s_1'$ and $s_1' \approx s_2'$.*

**Lemma 8.20** (Backward Matching Final States)**.** *Let $s_1$ and $s_2$ be dynamic states of* Inca *and* Dyn *respectively. If $s_1 \approx s_2$ and* $\mathsf{final}_{\text{Ubx}}\ s_2$*, then* $\mathsf{final}_{\text{Inca}}\ s_1$*.*

Finally, we prove a bisimulation between Dyn and Inca.

**Theorem 8.21** (Bisimulation)**.** *There is a bisimulation between* Inca *and* Ubx *w.r.t. $\approx$.*

*Proof Sketch.* By Lemmas 8.17 to 8.20. □

Theorem 8.21 corresponds to an interpretation of the locale bisimulation from Chapter 6.

### 8.6.3 Compilation from Inca to Ubx

We present a simple exemplary compilation function composed on three steps:

1. Lift the program from Inca to Ubx.
2. Optimize the program by using as many Ubx instructions as possible.
3. Ensure that the result is valid with respect to the abstract interpretation.

The optimization pass is based on an oracle—an abstract function of type $'fun \Rightarrow \mathbb{N} \Rightarrow type\ option$—which, given the position of a Load instruction in a function, evaluates to the expected unboxed type of the loaded value. A variant of the abstract interpretation used for the simulation relation optimizes instructions in a linear pass based on the following type information.

1. All function parameters have boxed dynamic types.
2. The type produced by Push is provided by inspecting the constant.
3. The type produced by Load is provided by the oracle, or assumed to be a boxed dynamic type if the oracle evaluates to †.
4. The type consumed by Store is obtained from the abstract interpretation.
5. The types consumed and produced by Op, OpInl, and Call are always boxed dynamic and their number depends on the arity of the operation or function.
6. The types consumed and produced by OpUbx is obtained from $typeof_{ubx}$.

This information provided by the oracle could either be given directly by the programmer or be the result of automatic run-time instrumentation. In the second case, the virtual machine would first execute code in Inca mode and gather some

statistics on encountered types, a stage usually referred to as *profiling*. When some heuristics indicate that a point of "dynamic locality of type usage" [7] is reached, the program would then be compiled to Ubx, and the control flow diverted to Ubx's execution engine.

The accuracy of the oracle's predictions may increase or decrease run-time performance, but may never alter the semantics of the executed program. If a value loaded from memory does not match the oracle's prediction, rule Load-Ubx-Miss generalizes the function back to cope with boxed values before resuming the execution.

We proved that compiled, optimized programs simulate their uncompiled counterparts.

**Lemma 8.22** (Compiled Matching States)**.** *Let $p_1$ and $p_2$ be static program representations of* Inca *and* Ubx *respectively. Let $s_1$ be a dynamic state of* Inca*. If* $\mathsf{compile}\, p_1 = p_2$ *and* $\mathsf{load}_{\text{Inca}}\, p_1\, s_1$*, then there exists a state $s_2$ such that* $\mathsf{load}_{\text{Ubx}}\, p_2\, s_2$ *and $s_1 \approx s_2$.*

**Theorem 8.23** (Compilation Function)**.** *The function* compile *is a compiler from* Inca *to* Ubx*, which guarantees that the* Ubx *program simulates the* Inca *program.*

*Proof Sketch.* By Lemmas 8.19, 8.20 and 8.22. □

Theorem 8.23 corresponds to an interpretation of the locale compiler from Chapter 6. The simulation framework gives us a proof that the successful execution of a compiled Ubx program has a behaviour equivalent to a behaviour of the original Inca program.

**Corollary 8.24** (Soundness of Compilation)**.** *Let $p_1$ and $p_2$ be static program representations of* Inca *and* Ubx *respectively. If $p_1$ compiles to $p_2$ (i.e.,* $\mathsf{compile}\, p_1 = p_2$*), and $p_2$ has behaviour $b_2$ (i.e., $p_2 \Downarrow b_2$), and $b_2$ is not a wrong behaviour, then $p_1$ has a behaviour that matches with $b_2$ (i.e., $\exists b_1.\ p_1 \Downarrow b_1 \wedge \mathsf{rel\_behaviour}\ () \approx b_1\ b_2$).*

Compilation from Inca to Ubx is incomplete, in the sense of Theorem 8.11, for two reasons.

First, the abstract interpretation and optimization passes are too simplistic to handle wrong type predictions from the oracle; this could be addressed by implementing and verifying more sophisticated passes.

Second, the hypothesis of Theorem 8.11 is too weak because loading a program does not guarantee that it can be successfully executed; the compilation pass, however, is dependent on a conservative abstract interpretation that assumes a successful execution. To address this issue and prove completeness, we would need to either strengthen the hypothesis (e.g., with a typing judgment that guarantees a valid execution) or totalize the semantics of the languages to guarantee that all programs have a valid execution (e.g., by adding exception handling to the semantics).

## 8.7 Practical Perspective

**Benefits of Formalization** The full-fledged CPython prototype successfully passed all relevant unit tests and ran major Python applications, benchmarks, and frameworks. Although tests covered several ten thousands lines of Python programs and

C code for libraries, some "Heisenbugs" occurred every now and then. Through the presented formalization, we were able to discern a new requirement that addressed the bug.

The new requirement—obvious in hindsight, but nonobvious before—is due to the deoptimization of Ubx optimized code. When deoptimizing a certain function `f`, a prior, yet incomplete call to function `f` may still be active on stack. Assume the prior stack frame of `f` was type-specialized to a specific type $T$ and that the operand stack of the interpreter stack frame contained unboxed data of type $T$. If we deoptimize the newer stack frame of the present function invocation of `f`, then all unboxed data will be boxed again and stored in memory. Now, assume that during a following call of function `f`, it will be optimized again, but to a different type $T'$. The program continues, until it eventually continues to operate on the prior stack frame belonging to function `f`. The interpreter operand stack may now hold unboxed data of type $T$, but the optimized instructions will assume the data to be of type $T'$. Potential errors following from this situation are: (i) deoptimization may fail when the types $T$ and $T'$ differ; (ii) execution of machine-native operations may fail, when the data representation differs; (iii) (un-)boxing of data may fail, when we try to access machine-native data incorrectly.

The underlying problem is that there is only one optimized interpreter code image stored for each interpreted function. A Ubx function is, therefore, not able to infer potential changes to its code. A variety of techniques address this issue (e.g., deoptimizing *all* invocations of the optimized code or keeping a version counter of the code image and check, that these are identical).

## 8.8 Related Work

To the best of our knowledge, there exists no prior work that is directly related to the formalization and verification of the speculative optimizations presented here. We therefore group the related work into the three most directly related groups of related work: (i) formalization and verification of translators, (ii) formalization and verification of dynamic languages, and (iii) just-in-time compiler optimizations.

### 8.8.1 Formalization and Verification of Translators

We combine the related work on compilers, just-in-time compilers, and interpreters and subsume all of them under the label "translators." From a historical perspective, the correctness of translators has been an active research area since at least the 1980s. The topic of compiler correctness has, for instance, been examined in the European FP2 research program ProCoS [63]. The findings of ProCoS subsequently lead to a larger German research project called Verifix, which examined several aspects of compiler correctness [56, 57]. In the 2000s, a group of researchers in France pioneered the field by mechanizing correctness of an industrial-strength C compiler [22, 74, 93, 114, 115, 116]. In the 2010s, a mechanized formalization and verification of ML followed [71].

In 2006, Klein and Nipkow formalized Jinja, a unified model of a Java-like source language, virtual machine, and compiler [67, 68]. Lochbihler later added support for interleaved execution of threads with JinjaThreads [75, 76, 77, 78, 79]. In 2018, Watt

mechanized the WebAssembly specification [121, 122].

In a similar vein, the verification of compile-time optimizations has received considerable attention from the research community. VellVM, e.g., focused on verifying optimizations on the LLVM bytecode intermediate representation [128]. Tatlock and Lerner simplify the verification of optimizations in verified compilers by using SMT solvers to aid with the construction of verified translation validators [109]. Prior research also focused on the formalization verification of intermediate representations, such as Java bytecode, without optimizations [73, 104].

In 2010, Myreen presents his work on the formalization and verification of just-in-time compilers [84], documenting some of the difficulties posed by self-modifying code. This paper is most directly related prior work, but addresses a different direction, namely, the formalization of just-in-time compilers. Our work, however, sidesteps the intricate difficulties of just-in-time compilers by focusing on optimizing interpreters instead. In 2017, Flückiger et al. investigated the correctness of speculative optimizations with dynamic deoptimization [53]. Since our virtual machine interpreters can be thought of as intermediate representations, the Inca language confirms the finding by Flückiger et al., namely that reasoning about complex system interactions is a lot easier by embedding the proper information in it. Moreover, Ubx goes further than Flückiger et al. by covering different data representations.

### 8.8.2 Formalization and Verification of Dynamic Languages

The formalization of dynamic languages in general, and JavaScript in particular, has been the subject of substantial prior work. In 2010, $\lambda_{JS}$ presented the first executable, formal semantics of JavaScript [61]. By rewriting JavaScript surface syntax into equivalent Scheme code, JavaScript programs could be executed, with correctness and security guarantees depending on the underlying Scheme system. In 2013, $\lambda_{\pi}$ applied a similar technique to provide a formal semantics for Python [92]. A comprehensive formalization and verification effort of JavaScript is the Coq-based project JSCert [23]. JSCert generates a verified JavaScript interpreter from its formalization.

While a formal semantics is an indispensable prerequisite for a correct and verified virtual machine, it addresses the desirable performance aspect insufficiently. To attain performance, a formalization of speculative optimizations is required, which is the key contribution of our chapter.

### 8.8.3 Just-in-Time Compilers and Interpreters

Aycock gives a good overview of the history of just-in-time compilers up until the early 2000s [1]. Particularly relevant prior work is the original work by Deutsch and Schiffman, which introduced the seminal idea of inline caching [49]. Originally, their work on Smalltalk 80 systems considered so-called *monomorphic* inline caches, i.e., inline caches that hold at most one address. Hölzle, Chambers and Ungar subsequently extended these with so-called polymorphic inline caches, i.e., a combination of an inline cache and a stub to cache multiple target addresses, which is particularly relevant in highly polymorphic call sites [64, 65].

In 1996, Roemer et al. studied the performance of interpreters and found no specific evidence to identify hints [95]. In 2003, Ertl and Gregg investigated the

performance of interpreters again and found evidence of the importance of branch predictors [51]. In 2009, Brunthaler analyzed the varying performance potential of interpreter optimizations and found that the interpreter abstraction level is the primary performance determinant for selecting interpreter optimizations [30]. In 2010, Brunthaler investigated the use of inline caching in a purely interpretative fashion, in contrast to its use in just-in-time compilers, and found speedups by a factor of up to 2 [31, 32]. In 2012, Würthinger et al. generalized Brunthaler's bytecode interpreter optimizations to abstract-syntax tree interpreters [126], which subsequently became the cornerstone for the development of the Truffle/Graal virtual machine implementation efforts [124, 125]. In 2014, Wang, Wu, and Padua demonstrated the potential of combining advanced optimizations in the R programming language and reported speedups of up to 3.5 [120].

All prior work in this area reports important speedups, either through dynamic code generation in a classic just-in-time compiler setting, or by way of optimizing interpreters. The exclusive focus of prior work is on improving performance, or sometimes also reducing memory footprint. The aspect of formalization and verification, in particular to establish correctness, is notably absent.

## 8.9 Conclusion

We presented a formalization of virtual machine interpreters for dynamically typed programming languages. Our formalization define an interpreter supporting the most representative features found and used by many virtual machine interpreters for mainstream languages. We then methodically extend the virtual machine interpreter's instruction set and semantics to accommodate increasingly specialized and optimized instruction derivatives. These incrementally specialized derivatives eliminate much of the overhead frequently found in high abstraction-level virtual machines, such as those used by Python or JavaScript.

The optimized instruction derivatives, in particular, first eliminate the overhead of dynamic typing by inline caching a prior recorded type at its place. This recorded type information is subsequently used to expand the local knowledge of type usage in a specific region of the program (e.g., a loop or a basic block). Once a suitable region of known types is determined, we can rewrite the whole sequence to eliminate the overhead of using boxed objects by using machine-native data representation instead.

Our formalization enables the proof of both soundness and completeness for speculative optimizations. Given a formal semantics of a dynamic language, and a suitable intermediate representation, our formalization provides a systematic way to (i) integrate speculative optimizations, and (ii) establish the correctness of the resulting system. We believe that our formalization provides a foundation for the verification of industrial-strength implementations. These implementations will benefit from our formalization's ability to pinpoint subtle errors and nonobvious requirements.

# Chapter 9

# Conclusion

This thesis described the formalizations of multiple results in the fields of automated reasoning and virtual machines. All formalizations were carried on in the Isabelle/HOL proof assistant. The results were separated in three parts.

**Part I: Correctness of Logical Calculi** The first presented formalization was of the ground superposition calculus. The main theorems formalized are soundness (i.e., every clause derived from valid clauses is valid) and static refutational completeness (i.e., if a saturated clause set is unsatisfiable, then it contains the empty clause). The proof of refutational completeness involves constructing a model for a satisfiable clause set and proving that this model construction will succeed if the clause set is saturated.

The second presented formalization was of the SCL(FOL) calculus. Compared with the pen-and-paper version, the formalized calculus is simpler and more general. The main theorems formalized are soundness (i.e., when the calculus stops, it either found a model or derived a contradiction), nonredundancy of learned clauses (i.e., learned clauses are not entailed by smaller known clauses), and termination (i.e., a run of the calculus will always terminate). The proof of termination involves a novel monotonically decreasing measuring function. Some formalized theorems are also stronger than their pen-and-paper versions. The formalization also led to finding and fixing one bug in a previously published version of the Backtrack rule.

**Part II: Simulations between Calculi** We first presented our framework for simulation proofs. It was originally developed to prove the correctness of compilation functions, but it turned out to be also useful for simulation proofs between logical calculi.

We then presented our formalization of a proof that SCL(FOL) can simulate nonredundant clause learning by ground ordered resolution. For this, we first formalized the ground ordered resolution calculus by adapting our formalization of ground superposition. The simulation proof itself involves defining appropriate strategies for both calculi. This result is not new, but we devised an alternative proof optimized for formalization, significantly simplified the strategy for SCL(FOL), and proved that the simulation also holds the other way around. Our new proof involves ten refinement steps. To simplify the formalization of these steps, we proved a lemma

| Chapter | Entry | Nonblank lines |
|---|---|---|
| 3 | `Abstract_Substitution` | 613 |
| 3 | `Min_Max_Least_Greatest` | 1 119 |
| 4 | `Superposition_Calculus` | 11 213 |
| | *Ground theories: 2964 nonblank lines* | |
| | *Nonground theories: 6723 nonblank lines* | |
| | *Other theories: 1526 nonblank lines* | |
| 5 | `Simple_Clause_Learning` | 9 138 |
| 6 | `VeriComp` | 1 670 |
| 7 | `SCL_Simulates_Ground_Resolution` | 25 251 |
| 8 | `Interpreter_Optimizations` | 6 542 |
| | | 55 546 |

Figure 9.1: Size of the discussed entries of the *Archive of Formal Proofs* version 2024-11-04.

that lifts a simulation to a bisimulation (i.e., a simulation in both directions) under some conditions.

**Part III: Simulations between Virtual Machines** We presented our formalization of three stack-based interpreter for dynamically typed bytecode languages. The first interpreter serves as a baseline and has features representative of real-world virtual machines. The second interpreter adds a speculative optimization known as inline caching that rewrites the program under execution at run time to use more efficient operations. The third interpreter adds another speculative optimization that allows to manipulate machine-native data and rewrite the program under execution if the optimization would not work. The main theorems formalized are that the first interpreter can simulate the second one and vice versa, and also that the second interpreter can simulate the third one and vice versa. Other formalized theorems include some completeness results of compilation from the first to the second interpreter and from the second to the third. This formalization was the initial use case for which the framework for simulation proofs was developed. The formalization also led us to discover the source of one bug in the original CPython prototype of the speculative optimizations.

These formalizations were carried out over several years and amount to over 55 000 nonblank lines[1] (Fig. 9.1). I wrote all of them except for the entry `Superposition_Calculus` that is joint work with Balazs Toth. A conservative estimate would put my contribution at over 47 000 nonblank lines plus some more in the Isabelle/HOL distribution or other entries of the *Archive of Formal Proofs*. I produced many reusable definitions and lemmas as byproduct and made them available to the whole Isabelle community.

These formalizations highlight the benefits of formalizing pen-and-paper proofs. Some formalizations led to the discovery of errors in previous proofs or software

[1] Counted using `grep -Ev '^[[:blank:]]*$'`.

prototypes. Some formalizations revealed missing assumptions; other revealed that some assumptions were unnecessary. Some formalizations lead to simplifications and generalizations; others lead to new interesting proofs of the same results. But the biggest benefit is probably that a formalization forces us to provide clear definitions, assumptions, and write detailed proofs. Pen-and-paper proofs have a tendency to contain vaguenesses that are adequate to convey an intuition to the reader, but inadequate for precise understanding. A significant portion of the formalization time is thus spent trying to understand what the pen-and-paper proofs meant and to rediscover the proof. For that reason, I am convinced that the best time to formalize a theory is when developing it for the first time. Another advantage is that a paper backed by a formalization can chose to omit some or all proofs and use the gained space to more clearly present and explain the concepts and lemmas. A reader interested in the proofs can then refer directly to the formalization.

# Appendix A

# Where to Find the Formalized Theorems?

This chapter presents where each lemma, theorem, and corollary presented in this thesis can be found in the different Isabelle/HOL formalizations. The references to the entries of the *Archive of Formal Proofs* may also contain the specific version to look at.

The formalization described in Chapter 4 is available in the entry `Superposition_Calculus` [47] of the *Archive of Formal Proofs* version 2024-11-04.

**Theorem 4.1** is for presentation purpose and has no correspondence in the formalization.

**Lemma 4.10** corresponds to
lemma termination_Union_rewrite_sys of
theory Ground_Superposition_Completeness.

**Lemma 4.11** corresponds to
lemma WCR_Union_rewrite_sys of
theory Ground_Superposition_Completeness.

**Lemma 4.12** is not explicitly stated but inlined directly in the proof of
lemma statically_complete of
theory Ground_Superposition_Completeness.

**Lemma 4.13** corresponds to
lemma true_cls_if_productive_epsilon and
lemma false_cls_if_productive_epsilon of
theory Ground_Superposition_Completeness.

**Lemma 4.14** corresponds to
lemma lift_entailment_to_Union of
theory Ground_Superposition_Completeness.

**Lemma 4.15** corresponds to
lemma model_preconstruction of
theory Ground_Superposition_Completeness.

**Lemma 4.16** corresponds to
lemma model_construction of
theory Ground_Superposition_Completeness.

**Theorem 4.17** corresponds to
lemma statically_complete of
theory Ground_Superposition_Completeness.

The formalization described in Chapter 5 is available in the entry `Simple_Clause_Learning` [40] of the *Archive of Formal Proofs* version 2024-11-04.

**Lemma 5.1** corresponds to
lemma scl_state_invariants of
theory Invariants.

**Lemma 5.5** corresponds to
lemma strategy_restrictions of
theory SCL_FOL.

**Theorem 5.6** corresponds to
lemma monotonicity_wrt_bound of
theory SCL_FOL.

**Theorem 5.7** corresponds to
lemma correct_termination of
theory Correct_Termination.

**Corollary 5.8** corresponds to
lemma correct_termination_strategies of
theory Correct_Termination.

**Theorem 5.11** corresponds to
lemma dynamic_non_redundancy_regular_scl of
theory Non_Redundancy.

**Corollary 5.12** corresponds to
lemma static_non_subsumption_regular_scl of
theory Non_Redundancy.

**Corollary 5.13** corresponds to
lemma dynamic_non_redundancy_strategy of
theory Non_Redundancy.

**Theorem 5.16** corresponds to
lemma termination_scl_without_back of
theory Termination.

**Lemma 5.19** corresponds to
lemma $\mathcal{M}$_back_after_regular_backtrack of
theory Termination.

**Theorem 5.20** corresponds to
lemma termination_regular_scl of
theory Termination.

**Corollary 5.21** corresponds to
lemma termination_strategy of
theory Termination.

**Theorem 5.22** corresponds to
lemma completeness_wrt_bound of
theory Completeness.

**Lemma 5.23** corresponds to
lemma ex_bound_if_unsat of
theory Completeness.

The formalization described in Chapter 6 is available in the entry `VeriComp` [38] of the *Archive of Formal Proofs* version 2024-11-04.

**Lemma 6.1** corresponds to
lemma right_unique_state_behaves of
theory Semantics.

**Lemma 6.2** corresponds to
lemma right_unique_prog_behaves of
theory Language.

**Theorem 6.3** corresponds to
lemma simulation_behaviour of
theory Simulation.

**Corollary 6.4** corresponds to
lemma behaviour_preservation of
theory Compiler.

**Lemma 6.5** corresponds to
lemma backward_simulation_composition of
theory Simulation.

**Theorem 6.6** corresponds to
lemma compiler_composition of
theory Simulation.

The formalization described in Chapter 7 is mostly available in the entries `VeriComp` [38] and `SCL_Simulates_Ground_Resolution` [43] of the *Archive of Formal Proofs* version 2024-11-04. Lemmas 7.10 and 7.11 are exceptions: They were added directly to the development version of the *Archive of Formal Proofs* (changeset 3fd35e3227256ed3) and should be available in the next public release. These lemmas are for presentation only and not used in the formalization of the calculi or refinement steps.

**Lemma 7.1** corresponds to
lemma unique_ground_resolution and
lemma unique_ground_factoring of
theory Background [43].

**Lemma 7.7** corresponds to
lemma lift_strong_simulation_to_bisimulation of
theory Lifting_Simulation_To_Bisimulation [38].

**Lemma 7.10** corresponds to
lemma final_iff_stuck_if_invar of
theory Lifting_Simulation_To_Bisimulation [38].

**Lemma 7.11** corresponds to
lemma wellbehaved_transition_systems_agree_on_final_iff_agree_on_stuck of
theory Lifting_Simulation_To_Bisimulation [38].

**Lemma 7.17** is not explicitly stated but corresponds to

lemma right_unique_ord_res_1 and
lemma ord_res_1_safe and
sublocale declaration ord_res_1_semantics of
theory ORD_RES_1 [43].

**Theorem 7.18** corresponds to
sublocale declaration backward_simulation_with_measuring_function of
theory Simulation_SCLFOL_ORDRES [43].

**Lemma 7.23** is not explicitly stated but corresponds to
lemma right_unique_ord_res_2_step and
lemma ord_res_2_step_safe and
sublocale declaration ord_res_2_semantics of
theory ORD_RES_2 [43].

**Lemma 7.25** corresponds to
lemma ord_res_1_final_iff_ord_res_2_final of
theory Simulation_SCLFOL_ORDRES [43].

**Lemma 7.26** corresponds to
lemma forward_simulation of
theory Simulation_SCLFOL_ORDRES [43].

**Lemma 7.27** corresponds to
lemma bisimulation_ord_res_1_ord_res_2 of
theory Simulation_SCLFOL_ORDRES [43].

**Theorem 7.35** corresponds to
lemma forward_simulation_ord_res_1_ord_res_11 and
lemma backward_simulation_ord_res_1_ord_res_11 of
theory Simulation_SCLFOL_ORDRES [43].

**Theorem 7.37** corresponds to
lemma ord_res_11_is_strategy_for_regular_scl of
theory Simulation_SCLFOL_ORDRES [43].

**Corollary 7.38** corresponds to
lemma ord_res_11_non_subsumption of
theory Simulation_SCLFOL_ORDRES [43].

**Corollary 7.39** corresponds to
lemma ord_res_11_termination of
theory Simulation_SCLFOL_ORDRES [43].

The formalization described in Chapter 8 is available in the entries `VeriComp` [38] and `Interpreter_Optimizations` [39] of the *Archive of Formal Proofs* version 2021-02-23. Later versions of the *Archive of Formal Proofs* contain a refactored and extended version of the formalization.

**Lemma 8.3** corresponds to
lemma forward_lockstep_simulation of
theory Std_to_Inca_simulation [39].

**Lemma 8.4** corresponds to
lemma match_final_forward of
theory Std_to_Inca_simulation [39].

**Lemma 8.5** corresponds to
lemma backward_lockstep_simulation of
theory Std_to_Inca_simulation [39].

**Lemma 8.6** corresponds to
lemma match_final_backward of
theory Std_to_Inca_simulation [39].

**Theorem 8.7** corresponds to
sublocale declaration std_inca_bisimulation of
theory Std_to_Inca_simulation [39].

**Lemma 8.8** corresponds to
lemma compile_load of
theory Std_to_Inca_compiler [39].

**Theorem 8.9** corresponds to
sublocale declaration std_to_inca_compiler of
theory Std_to_Inca_compiler [39].

**Corollary 8.10** is not explicitly stated but corresponds to the instantiation of
lemma behaviour_preservation of
theory Compiler [38].

**Theorem 8.11** corresponds to
lemma compile_load_forward of
theory Std_to_Inca_compiler [39].

**Lemma 8.17** corresponds to
lemma forward_lockstep_simulation of
theory Inca_to_Ubx_simulation [39].

**Lemma 8.18** corresponds to
lemma match_final_forward of
theory Inca_to_Ubx_simulation [39].

**Lemma 8.19** corresponds to
lemma backward_lockstep_simulation of
theory Inca_to_Ubx_simulation [39].

**Lemma 8.20** corresponds to
lemma match_final_backward of
theory Inca_to_Ubx_simulation [39].

**Theorem 8.21** corresponds to
sublocale declaration inca_ubx_bisimulation of
theory Inca_to_Ubx_simulation [39].

**Lemma 8.22** corresponds to
lemma compile_load of
theory Inca_to_Ubx_compiler [39].

**Theorem 8.23** corresponds to
sublocale declaration std_to_inca_compiler of
theory Inca_to_Ubx_compiler [39].

**Corollary 8.24** is not explicitly stated but corresponds to the instantiation of
lemma behaviour_preservation of

theory Compiler [38].

# Bibliography

[1] John Aycock. "A brief history of just-in-time." In: *ACM Computing Surveys* 35.2 (2003), pages 97–113. issn: 0360-0300. doi: http://doi.acm.org/10.1145/857076.857077 (cited on page 129).

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. 1st edition. Cambridge, United Kingdom: Cambridge University Press, 1998. isbn: 978-0-521-77920-3. doi: 10.1017/CBO9781139172752 (cited on pages 19, 32, 36, 42).

[3] Harald Bachmair Leo an Ganzinger. "Resolution Theorem Proving." In: *Handbook of Automated Reasoning*. Edited by Alan Robinson and Andrei Voronkov. Volume I. Elsevier and MIT Press, 2001, pages 19–99. isbn: 978-0-444-82949-8 (cited on pages 26, 29).

[4] Leo Bachmair and Harald Ganzinger. "On Restrictions of Ordered Paramodulation with Simplification." In: *CADE-10*. Edited by Mark E. Stickel. Volume 449. Lecture Notes in Computer Science. Springer, 1990, pages 427–441. doi: 10.1007/3-540-52885-7_105 (cited on pages 5, 25, 27, 29).

[5] Leo Bachmair and Harald Ganzinger. "Rewrite-based Equational Theorem Proving with Selection and Simplification." In: *Journal of Logic and Computation* 4.3 (June 1994), pages 217–247. issn: 0955-792X. doi: 10.1093/logcom/4.3.217 (cited on pages 5, 25–27, 29, 30).

[6] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. "Theorem Proving for Hierarchic First-Order Theories." In: *ALP '92*. Edited by Hélène Kirchner and Giorgio Levi. Volume 632. Lecture Notes in Computer Science. Springer, 1992, pages 420–434. doi: 10.1007/BFB0013841 (cited on page 40).

[7] Scott B. Baden. *High Performance Storage Reclamation in an Object-Based Memory System*. Technical report. Berkeley, CA, USA: University of California, Berkeley, 1982 (cited on pages 104, 127).

[8] Clemens Ballarin. "Locales. A Module System for Mathematical Theories." In: *Journal of Automated Reasoning* 52.2 (February 2014), pages 123–153. issn: 0168-7433. doi: 10.1007/s10817-013-9284-7 (cited on pages 6, 13, 31, 42, 62, 106).

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification. 23rd International Conference*. CAV 2011 (Snowbird, UT, USA, 14 July 2011–20 July 2011). Edited by Ganesh Gopalakrishnan and Shaz Qadeer. Volume 6806. Lecture Notes in Computer Science. Springer

Berlin Heidelberg, 5 July 2011, pages 171–177. isbn: 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_14 (cited on page 12).

[10] Peter Baumgartner, Joshua Bax, and Uwe Waldmann. "Beagle. A Hierarchic Superposition Theorem Prover." In: *CADE 25*. Edited by Amy P. Felty and Aart Middeldorp. Volume 9195. Lecture Notes in Computer Science. Springer, 2015, pages 367–377. doi: 10.1007/978-3-319-21401-6_25 (cited on pages 5, 25).

[11] Peter Baumgartner and Uwe Waldmann. "Hierarchic Superposition Revisited." In: *Description Logic, Theory Combination, and All That—Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*. Edited by Carsten Lutz, Uli Sattler, Cesare Tinelli, Anni-Yasmin Turhan, and Frank Wolter. Volume 11560. Lecture Notes in Computer Science. Springer, 2019, pages 15–56. doi: 10.1007/978-3-030-22102-7_2 (cited on page 40).

[12] Roberto J. Bayardo and Robert Schrag. "Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances." In: *Principles and Practice of Constraint Programming — CP'96. Second International Conference*. Edited by Eugene C. Freuder. Volume 1118. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 7 August 1996, pages 46–60. isbn: 978-3-540-61551-4. doi: 10.1007/3-540-61551-2_65 (cited on pages 5, 41, 71).

[13] Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, and Uwe Waldmann. "Superposition for Lambda-Free Higher-Order Logic." In: *Logical Methods in Computer Science* 17.2 (2021-04-12). doi: 10.23638/LMCS-17(2:1)2021 (cited on pages 26, 38).

[14] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. "Superposition for Higher-Order Logic." In: *Journal of Automated Reasoning* 67.10 (2023). issn: 0168-7433. doi: 10.1007/S10817-022-09649-9 (cited on pages 12, 25, 26, 38, 40).

[15] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. "Superposition with Lambdas." In: *Journal of Automated Reasoning* 65.7 (2021), pages 893–940. issn: 0168-7433. doi: 10.1007/S10817-021-09595-Y (cited on pages 26, 38).

[16] Ahmed Bhayat and Giles Reger. "A Combinator-Based Superposition Calculus for Higher-Order Logic." In: *Automated Reasoning. Proceedings of the 10th International Joint Conference, Part I*. IJCAR 2020 (Paris, France, 1 July 2020–4 July 2020). Edited by Nicolas Peltier and Viorica Sofronie-Stokkermans. Volume 12166. Lecture Notes in Computer Science. Springer International Publishing, 30 June 2020, pages 278–296. isbn: 978-3-030-51073-2. doi: 10.1007/978-3-030-51074-9_16 (cited on pages 12, 25, 26, 40).

[17] Ahmed Bhayat, Johannes Schoisswohl, and Michael Rawson. "Superposition with Delayed Unification." In: *Automated Deduction – CADE 29. Proceedings of the 29th International Conference on Automated Deduction*. CADE 2023 (Rome, Italy, 1 July 2023–4 July 2023). Edited by Brigitte Pientka and Cesare Tinelli. Volume 14132. Lecture Notes in Computer Science. Springer Cham, 3 September 2023, pages 23–40. isbn: 978-3-031-38498-1. doi: 10.1007/978-3-031-38499-8_2 (cited on pages 26, 38).

[18] Jasmin Christian Blanchette. "Formalizing the Metatheory of Logical Calculi and Automatic Provers in Isabelle/HOL (Invited Talk)." In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019 (Cascais, Portugal, 14 January 2019–15 January 2019). Edited by Assia Mahboubi and Magnus O. Myreen. Association for Computing Machinery, 14 January 2019, pages 1–13. isbn: 978-1-4503-6222-1. doi: 10.1145/3293880.3294087. The IsaFoL (Isabelle Formalization of Logic) repository is hosted at https://github.com/IsaFoL/IsaFoL (cited on pages 27, 39, 41, 73).

[19] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality." In: *Journal of Automated Reasoning* 61.1 (June 2018), pages 333–365. issn: 0168-7433. doi: 10.1007/s10817-018-9455-7 (cited on pages 38, 42).

[20] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality." In: *Automated Reasoning. Proceedings of the 8th International Joint Conference*. IJCAR 2016 (Coimbra, Portugal, 26 June 2016–2 July 2016). Edited by Nicola Olivetti and Ashish Tiwari. Volume 9706. Lecture Notes in Computer Science. Springer Cham, 12 June 2016, pages 25–44. isbn: 978-3-319-40228-4. doi: 10.1007/978-3-319-40229-1_4 (cited on page 42).

[21] Jasmin Christian Blanchette and Sophie Tourret. "Extensions to the Comprehensive Framework for Saturation Theorem Proving." In: *Archive of Formal Proofs* (August 2020). https://isa-afp.org/entries/Saturation_Framework_Extensions.html, Formal proof development. issn: 2150-914x (cited on pages 5, 30, 34, 42).

[22] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End." In: *FM 2006: Formal Methods. 14th International Symposium on Formal Methods*. FM 2006 (Hamilton, Canada, 21 August 2006–17 August 2006). Edited by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Volume 4085. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 10 August 2006, pages 460–475. isbn: 978-3-540-37215-8. doi: 10.1007/11813040_31 (cited on page 128).

[23] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. "A Trusted Mechanised JavaScript Specification." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: Association for Computing Machinery, 2014, pages 87–100. isbn: 978-1-4503-2544-8. doi: 10.1145/2535838.2535876 (cited on page 129).

[24] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. "veriT. An Open, Trustable and Efficient SMT-Solver." In: *Automated Deduction – CADE 22. 22nd International Conference on Automated Deduction*. CADE 2009 (Montreal, Canada, 2 August 2009–7 August 2009). Edited by Renate A. Schmidt. Volume 5663. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 10 July 2009, pages 151–156. isbn: 978-3-642-02958-5. doi: 10.1007/978-3-642-02959-2_12 (cited on page 12).

[25] Martin Bromberger, Martin Desharnais, and Christoph Weidenbach. "An Isabelle/HOL Formalization of the SCL(FOL) Calculus." In: *Automated Deduction – CADE 29. Proceedings of the 29th International Conference on Automated Deduction*. CADE 2023 (Rome, Italy, 1 July 2023–4 July 2023). Edited by Brigitte Pientka and Cesare Tinelli. Volume 14132. Lecture Notes in Computer Science. Springer Cham, 3 September 2023, pages 116–133. isbn: 978-3-031-38498-1. doi: 10.1007/978-3-031-38499-8_7 (cited on pages 6, 19, 39, 41, 71, 79).

[26] Martin Bromberger, Alberto Fiori, and Christoph Weidenbach. "Deciding the Bernays-Schoenfinkel Fragment over Bounded Difference Constraints by Simple Clause Learning over Theories." In: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*. Edited by Fritz Henglein, Sharon Shoham, and Yakir Vizel. Volume 12597. Lecture Notes in Computer Science. Springer, 2021, pages 511–533. doi: 10.1007/978-3-030-67067-2_23 (cited on pages 5, 41, 53, 71).

[27] Martin Bromberger, Chaahat Jain, and Christoph Weidenbach. "SCL(FOL) Can Simulate Non-Redundant Superposition Clause Learning." In: *Automated Deduction – CADE 29. Proceedings of the 29th International Conference on Automated Deduction*. CADE 2023 (Rome, Italy, 1 July 2023–4 July 2023). Edited by Brigitte Pientka and Cesare Tinelli. Volume 14132. Lecture Notes in Computer Science. Springer Cham, 3 September 2023, pages 134–152. isbn: 978-3-031-38498-1. doi: 10.1007/978-3-031-38499-8_8 (cited on pages 6, 72, 73, 79, 97).

[28] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. "Exploring Partial Models with SCL." In: *Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, August, 11 - 12, 2022*. Edited by Boris Konev, Claudia Schon, and Alexander Steen. Volume 3201. CEUR Workshop Proceedings. CEUR-WS.org, 2022. url: http://ceur-ws.org/Vol-3201/paper5.pdf (cited on pages 5, 41, 71).

[29] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. *SCL(FOL) Revisited*. 2023. doi: 10.48550/arXiv.2302.05954. arXiv: 2302.05954 [cs.LO] (cited on pages 5, 41, 56, 71).

[30] Stefan Brunthaler. "Virtual-Machine Abstraction and Optimization Techniques." In: *Electronic Notes in Theoretical Computer Science* 253.5 (2009), pages 3–14 (cited on page 130).

[31] Stefan Brunthaler. "Efficient interpretation using quickening." In: *DLS '10. Proceedings of the 6th symposium on Dynamic languages*. DLS 2010 (Reno/Tahoe, Nevada, USA, 18 August 2010). Edited by William D. Clinger. New York, NY, USA: Association for Computing Machinery, 18 August 2010, pages 1–14. isbn: 978-1-4503-0405-4. doi: 10.1145/1869631.1869633 (cited on pages 7, 101, 130).

[32] Stefan Brunthaler. "Inline Caching Meets Quickening." In: *ECOOP 2010 – Object-Oriented Programming. 24th European Conference*. ECOOP 2010 (Maribor, Slovenia, 21 June 2010–25 June 2010). Edited by Theo D'Hondt. Volume 6183. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 17 June 2010, pages 429–451. isbn: 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_21 (cited on pages 7, 101, 130).

[33] Amine Chaieb and Tobias Nipkow. "Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic." In: *Logic for Programming, Artificial Intelligence, and Reasoning. 12th International Conference*. LPAR 2005 (Montego Bay, Jamaica, 2 December 2005–6 December 2005). Edited by Geoff Sutcliffe and Andrei Voronkov. Volume 3835. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 24 November 2005, pages 367–380. isbn: 978-3-540-30553-8. doi: 10.1007/11591191_26 (cited on pages 3, 12).

[34] Weidenbach Christoph, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. "SPASS version 3.5." In: *Automated Deduction – CADE 22. 22nd International Conference on Automated Deduction*. CADE 2009 (Montreal, Canada, 2 August 2009–7 August 2009). Edited by Renate A. Schmidt. Volume 5663. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 10 July 2009, pages 140–145. isbn: 978-3-642-02958-5. doi: 10.1007/978-3-642-02959-2_10 (cited on pages 5, 12, 25).

[35] Alonzo Church. "A formulation of the simple theory of types." In: *Journal of Symbolic Logic* 5.2 (2 June 1940). Edited by Alonzo Church and Ernest Nagel, pages 56–68. issn: 0022-4812. doi: 10.2307/2266170 (cited on page 9).

[36] Simon Cruanes. "Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond." Ph.D. thesis. École polytechnique, 2015 (cited on pages 5, 12, 25).

[37] Nachum Dershowitz and Zohar Manna. "Proving termination with multiset orderings." In: *Communications of the ACM* 22.8 (August 1979), pages 465–476. issn: 0001-0782. doi: 10.1145/359138.359142 (cited on page 23).

[38] Martin Desharnais. "A Generic Framework for Verified Compilers." In: *Archive of Formal Proofs* (February 2020). https://isa-afp.org/entries/VeriComp.html, Formal proof development. issn: 2150-914x (cited on pages 6, 62, 137–140).

[39] Martin Desharnais. "Inline Caching and Unboxing Optimization for Interpreters." In: *Archive of Formal Proofs* (December 2020). https://isa-afp.org/entries/Interpreter_Optimizations.html, Formal proof development. issn: 2150-914x (cited on pages 7, 102, 138, 139).

[40] Martin Desharnais. "A Formalization of the SCL(FOL) Calculus: Simple Clause Learning for First-Order Logic." In: *Archive of Formal Proofs* (April 2023). https://isa-afp.org/entries/Simple_Clause_Learning.html, Formal proof development. issn: 2150-914x (cited on pages 6, 41, 97, 136).

[41] Martin Desharnais. "Abstract Substitution." In: *Archive of Formal Proofs* (September 2024). https://isa-afp.org/entries/Abstract_Substitution.html, Formal proof development. issn: 2150-914x (cited on pages 5, 22).

[42] Martin Desharnais. "Minimal, Maximal, Least, and Greatest Elements w.r.t. Restricted Ordering." In: *Archive of Formal Proofs* (October 2024). https://isa-afp.org/entries/Min_Max_Least_Greatest.html, Formal proof development. issn: 2150-914x (cited on pages 5, 24).

[43] Martin Desharnais. "SCL Simulates Nonredundant Ground Resolution." In: *Archive of Formal Proofs* (October 2024). https://isa-afp.org/entries/SCL_Simulates_Ground_Resolution.html, Formal proof development. issn: 2150-914x (cited on pages 6, 73, 137, 138).

[44] Martin Desharnais and Stefan Brunthaler. "A Generic Framework for Verified Compilers Using Isabelle/HOL's Locales." In: *Isabelle Workshop*. Edited by Tobias Nipkow, Larry Paulson, and Makarius Wenzel. 2020. url: https://sketis.net/isabelle/isabelle-workshop-2020 (cited on pages 6, 61).

[45] Martin Desharnais and Stefan Brunthaler. "A Generic Framework for Verified Compilers Using Isabelle/HOL's Locales." In: *31ème Journées Francophones des Langages Applicatifs*. Edited by Zaynah Lea Dargaye and Yann Regis-Gianas. IRIF, January 2020. url: https://inria.hal.science/hal-02427360 (cited on pages 6, 61).

[46] Martin Desharnais and Stefan Brunthaler. "Towards Efficient and Verified Virtual Machines for Dynamic Languages." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021 (Virtual, Denmark, 17 January 2021–19 January 2021). Edited by Cătălin Hriţcu and Andrei Popescu. Association for Computing Machinery, 20 January 2021, pages 61–75. isbn: 978-1-4503-8299-1. doi: 10.1145/3437992.3439923 (cited on pages 7, 101).

[47] Martin Desharnais and Balazs Toth. "A Modular Formalization of Superposition." In: *Archive of Formal Proofs* (October 2024). https://isa-afp.org/entries/Superposition_Calculus.html, Formal proof development. issn: 2150-914x (cited on pages 5, 27, 135).

[48] Martin Desharnais, Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret. "A Modular Formalization of Superposition in Isabelle/HOL." In: *15th International Conference on Interactive Theorem Proving*. ITP 2024 (Tbilisi, Georgia, 9 September 2024–14 September 2024). Edited by Yves Bertot, Temur Kutsia, and Michael Norrish. Volume 309. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2 September 2024, 12:1–12:20. isbn: 978-3-95977-337-9. doi: 10.4230/LIPIcs.ITP.2024.12 (cited on pages 5, 19, 25).

[49] L. Peter Deutsch and Allan M. Schiffman. "Efficient Implementation of the Smalltalk-80 System." In: *POPL '84. Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL 1984 (Salt Lake City, UT, USA, 15 January 1984–18 January 1984). Edited by Ken Kennedy and Mary S. van Deusen. New York, NY, USA: Association for Computing Machinery, 15 January 1984, pages 297–302. isbn: 978-0-89791-125-2. doi: 10.1145/800017.800542 (cited on pages 104, 129).

[50] André Duarte and Konstantin Korovin. "Implementing Superposition in iProver (System Description)." In: *Automated Reasoning. Proceedings of the 10th International Joint Conference, Part II.* IJCAR 2020 (Paris, France, 1 July 2020–4 July 2020). Edited by Nicolas Peltier and Viorica Sofronie-Stokkermans. Volume 12166. Lecture Notes in Computer Science. Springer International Publishing, 30 June 2020, pages 388–397. isbn: 978-3-030-51053-4. doi: 10.1007/978-3-030-51054-1_24 (cited on pages 5, 25).

[51] M. Anton Ertl and David Gregg. "The Structure and Performance of Efficient Interpreters." In: *Journal of Instruction-Level Parallelism* 5 (November 2003). Edited by Eric Rotenberg. issn: 1942-9525. url: https://jilp.org/vol5/index.html (cited on pages 7, 101, 129, 130).

[52] Alberto Fiori and Christoph Weidenbach. "SCL Clause Learning from Simple Models." In: *Automated Deduction – CADE 27. Proceedings of the 27th International Conference on Automated Deduction.* CADE 2019 (Natal, Brazil, 27 August 2019–30 August 2019). Edited by Pascal Fontaine. Volume 11716. Lecture Notes in Computer Science. Springer Cham, 21 August 2019, pages 233–249. doi: 10.1007/978-3-030-29436-6_14 (cited on pages 5, 39, 41, 53, 71).

[53] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. "Correctness of Speculative Optimizations with Dynamic Deoptimization." In: *Proc. ACM Program. Lang.* 2.POPL (December 2017). doi: 10.1145/3158137 (cited on page 129).

[54] Asta Halkjær From, Patrick Blackburn, and Jørgen Villadsen. "Formalizing a Seligman-Style Tableau System for Hybrid Logic (Short Paper)." In: *Automated Reasoning. Proceedings of the 10th International Joint Conference, Part I.* IJCAR 2020 (Paris, France, 1 July 2020–4 July 2020). Edited by Nicolas Peltier and Viorica Sofronie-Stokkermans. Volume 12166. Lecture Notes in Computer Science. Springer International Publishing, 30 June 2020, pages 474–481. isbn: 978-3-030-51073-2. doi: 10.1007/978-3-030-51074-9_27 (cited on page 38).

[55] Asta Halkjær From, Anders Schlichtkrull, and Jørgen Villadsen. "A sequent calculus for first-order logic formalized in Isabelle/HOL." In: *Journal of Logic and Computation* 33.4 (April 2023), pages 818–836. issn: 0955-792X. doi: 10.1093/logcom/exad013 (cited on page 38).

[56] Sabine Glesner, G. Goos, F. v. Henke, H. Langmaack, W. Goerigk, and W. Zimmermann. *Abschlussbericht Verifix.* Technical Report. Universitäten Karlsruhe, Kiel, Ulm, July 2004 (cited on page 128).

[57] Gerhard Goos and Wolf Zimmermann. "Verification of Compilers." In: *Correct System Design: Recent Insights and Advances.* Edited by Ernst-Rüdiger Olderog and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pages 201–230. isbn: 978-3-540-48092-1. doi: 10.1007/3-540-48092-7_10 (cited on page 128).

[58] Samuel Groß. *JITSploitation I. A JIT Bug.* Google Project Zero. 1 September 2020. url: https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html (visited on 21 September 2020) (cited on page 101).

[59] Samuel Groß. *JITSploitation II. Getting Read/Write*. Google Project Zero. 1 September 2020. url: https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html (visited on 21 September 2020) (cited on page 101).

[60] Samuel Groß. *JITSploitation II. Subverting Control Flow*. Google Project Zero. 1 September 2020. url: https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html (visited on 21 September 2020) (cited on page 101).

[61] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of JavaScript." In: 2010, pages 126–150. doi: 10.1007/978-3-642-14107-2_7 (cited on page 129).

[62] John Harrison. "Formalizing Basic First Order Model Theory." In: *TPHOLs '98*. Edited by Jim Grundy and Malcolm C. Newey. Volume 1479. Lecture Notes in Computer Science. Springer, 1998, pages 153–170 (cited on page 39).

[63] Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, editors. *Provably Correct Systems*. NASA Monographs in Systems and Software Engineering. Springer, 2017. isbn: 978-3-319-48627-7. doi: 10.1007/978-3-319-48628-4 (cited on page 128).

[64] Urs Hölzle, Craig Chambers, and David M Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches." In: *Proceedings of the 5th European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991 (ECOOP'91)*. Volume 512/1991. Lecture Notes in Computer Science. Springer-Verlag, 1991, pages 21–38. isbn: 3-540-54262-0 (cited on page 129).

[65] Urs Hölzle and David Ungar. "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback." In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94*. New York, New York, USA: ACM Press, 1994, pages 326–336. isbn: 978-0-89791-662-2. doi: 10.1145/178243.178478 (cited on page 129).

[66] Gérard Huet and Derek C Oppen. "Equations and rewrite rules: A survey." In: *Formal Language Theory* (1980), pages 349–405 (cited on page 23).

[67] Gerwin Klein and Tobias Nipkow. "Jinja is not Java." In: *Archive of Formal Proofs* (June 2005). https://isa-afp.org/entries/Jinja.html, Formal proof development. issn: 2150-914x (cited on page 128).

[68] Gerwin Klein and Tobias Nipkow. "A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler." In: *ACM Trans. Program. Lang. Syst.* 28.4 (July 2006), pages 619–695. issn: 0164-0925. doi: 10.1145/1146809.1146811 (cited on page 128).

[69] Donald E. Knuth and Peter B. Bendix. "Simple word problems in universal algebras." In: *Computational Problems in Abstract Algebra*. Edited by John Leech. Pergamon Press, 1970, pages 263–297 (cited on pages 23, 42).

[70] Laura Kovács and Andrei Voronkov. "First-order theorem proving and Vampire." In: *CAV 2013*. Edited by Natasha Sharygina and Helmut Veith. Volume 8044. Lecture Notes in Computer Science. Springer, 2013, pages 1–35. doi: 10.1007/978-3-642-39799-8_1 (cited on pages 5, 12, 25).

[71] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML. A Verified Implementation of ML." In: *POPL '14. Proceedings of the 41th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL 2014 (San Diego, CA, USA, 22 January 2014–24 January 2014). Edited by Suresh Jagannathan and Peter Sewell. New York, NY, USA: Association for Computing Machinery, 11 January 2014, pages 179–191. isbn: 978-1-4503-2544-8. doi: 10.1145/2535838.2535841 (cited on pages 61, 128).

[72] Hendrik Leidinger and Christoph Weidenbach. "SCL(EQ): SCL for First-Order Logic with Equality." In: *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*. Edited by Jasmin Blanchette, Laura Kovács, and Dirk Pattinson. Volume 13385. Lecture Notes in Computer Science. Springer, 2022, pages 228–247. doi: 10.1007/978-3-031-10769-6_14 (cited on pages 5, 41, 71).

[73] Xavier Leroy. "Java Bytecode Verification: Algorithms and Formalizations." In: *Journal of Automated Reasoning* 30.3/4 (2003), pages 235–269. issn: 0168-7433. doi: 10.1023/A:1025055424017 (cited on page 129).

[74] Xavier Leroy. "A Formally Verified Compiler Back-end." In: *Journal of Automated Reasoning* 43.4 (December 2009), pages 363–446. issn: 0168-7433. doi: 10.1007/s10817-009-9155-4 (cited on pages 61, 101, 128).

[75] Andreas Lochbihler. "Jinja with Threads." In: *Archive of Formal Proofs* (December 2007). https://isa-afp.org/entries/JinjaThreads.html, Formal proof development. issn: 2150-914x (cited on page 128).

[76] Andreas Lochbihler. "Type Safe Nondeterminism - A Formal Semantics of Java Threads." In: *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*. January 2008. url: http://www.infsec.ethz.ch/people/andreloc/publications/lochbihler08fool.pdf (cited on page 128).

[77] Andreas Lochbihler. "Verifying a Compiler for Java Threads." In: *European Symposium on Programming (ESOP'10)*. Edited by A. D. Gordon. Volume 6012. Lecture Notes in Computer Science. Springer, March 2010, pages 427–447. doi: 10.1007/978-3-642-11957-6_23 (cited on page 128).

[78] Andreas Lochbihler. "Java and the Java Memory Model. A Unified, Machine-Checked Formalisation." In: *Programming Languages and Systems*. Edited by Helmut Seidl. Volume 7211. Lecture Notes in Computer Science. Springer, March 2012, pages 497–517. doi: 10.1007/978-3-642-28869-2_25 (cited on page 128).

[79] Andreas Lochbihler and Lukas Bulwahn. "Animating the Formalised Semantics of a Java-like Language." In: *Interactive Theorem Proving*. Edited by Marko van Eekelen, Herman Geuvers, Julien Schmalz, and Freek Wiedijk. Volume 6898. Lecture Notes in Computer Science. Springer, 2011, pages 216–232. doi: 10.1007/978-3-642-22863-6_17 (cited on page 128).

[80] Alexander Lochmann, Bertram Felgenhauer, Christian Sternagel, René Thiemann, and Thomas Sternagel. "Regular Tree Relations." In: *Archive of Formal Proofs* (2021). https://isa-afp.org/entries/Regular_Tree_Relations.html. issn: 2150-914x (cited on page 31).

[81] Piotr Łukowski. *Paradoxes*. Trends in Logic. Springer Dordrecht, 3 June 2011. isbn: 978-94-007-1475-5. doi: 10.1007/978-94-007-1476-2 (cited on page 2).

[82] João P. Marques Silva and Karem A. Sakallah. "GRASP-A new search algorithm for satisfiability." In: *Proceedings of International Conference on Computer Aided Design*. 1996, pages 220–227. doi: 10.1109/ICCAD.1996.569607 (cited on pages 5, 41, 71).

[83] Leonardo de Moura and Nikolaj Bjørner. "Z3. An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 337–340. isbn: 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24 (cited on page 12).

[84] Magnus O. Myreen. "Verified Just-In-Time Compiler on x86." In: *POPL '10. Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL 2010 (Madrid, Spain, 17 January 2010–23 January 2010). Edited by Jens Palsberg. New York, NY, USA: Association for Computing Machinery, 17 January 2010, pages 107–118. isbn: 978-1-60558-479-9. doi: 10.1145/1706299.1706313 (cited on pages 101, 129).

[85] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. 1st edition. Volume 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 3 April 2002. isbn: 978-3-540-43376-7. doi: 10.1007/3-540-45949-9 (cited on pages 3, 26, 61).

[86] Lawrence C. Paulson. "The Foundation of a Generic Theorem Prover." In: *Journal of Automated Reasoning* 5.3 (September 1989), pages 363–397. issn: 1573-0670. doi: 10.1007/BF00248324 (cited on page 9).

[87] Lawrence C. Paulson. "A Machine-Assisted Proof of Gödel's Incompleteness theorems for the Theory of Hereditarily Finite Sets." In: *Rev. Symb. Log.* 7.3 (2014), pages 484–498 (cited on page 39).

[88] Lawrence C. Paulson. "A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle." In: *Journal of Automated Reasoning* 55.1 (2015), pages 1–37. issn: 0168-7433. doi: 10.1007/S10817-015-9322-8 (cited on page 39).

[89] Lawrence C. Paulson and Jasmin Christian Blanchette. "Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers." In: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*. Edited by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Volume 2. EPiC Series in Computing. EasyChair, 2010, pages 1–11. doi: 10.29007/36dt (cited on pages 3, 12, 42).

[90] Nicolas Peltier. "A variant of the superposition calculus." In: *Archive of Formal Proofs* (2016). https://www.isa-afp.org/entries/SuperCalc.html (cited on pages 26, 39).

[91] Henrik Persson. "Constructive Completeness of Intuitionistic Predicate Logic: A Formalisation in Type Theory." Licentiate Thesis. Chalmers tekniska högskola and Göteborgs universitet, 1996 (cited on page 39).

[92] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. "Python. The Full Monty." In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 217–232. isbn: 978-1-4503-2374-1. doi: 10.1145/2509136.2509536 (cited on page 129).

[93] Silvain Rideau and Xavier Leroy. "Validating Register Allocation and Spilling." In: 2010, pages 224–243. doi: 10.1007/978-3-642-11970-5_13 (cited on page 128).

[94] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle." In: *Journal of the ACM* 12.1 (1965), pages 23–41. doi: 10.1145/321250.321253 (cited on page 26).

[95] Theodore H Romer, Dennis Lee, Geoffrey M Voelker, Alec Wolman, Wayne A Wong, Jean-Loup Baer, Brian N Bershad, and Henry M Levy. "The Structure and Performance of Interpreters." In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Association for Computing Machinery, 1996, pages 150–159. isbn: 978-0-89791-767-4. doi: 10.1145/248208.237175 (cited on page 129).

[96] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. isbn: 978-1-107-00363-7. doi: 10.1017/CBO9780511777110 (cited on page 62).

[97] Anders Schlichtkrull, Jasmin Blanchette, Dmitriy Traytel, and Uwe Waldmann. "Formalizing Bachmair and Ganzinger's Ordered Resolution Prover." In: *Journal of Automated Reasoning* 64.7 (October 2020), pages 1169–1195. issn: 0168-7433. doi: 10.1007/s10817-020-09561-0 (cited on pages 38, 42).

[98] Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. "A Verified Prover Based on Ordered Resolution." In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019 (Cascais, Portugal, 14 January 2019–15 January 2019). Edited by Assia Mahboubi and Magnus O. Myreen. Association for Computing Machinery, 14 January 2019, pages 152–165. isbn: 978-1-4503-6222-1. doi: 10.1145/3293880.3294100 (cited on page 38).

[99] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. "Formalization of Bachmair and Ganzinger's Ordered Resolution Prover." In: *Archive of Formal Proofs* (January 2018). https://isa-afp.org/entries/Ordered_Resolution_Prover.html, Formal proof development. issn: 2150-914x (cited on pages 5, 20, 22, 42).

[100] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. "Formalizing Bachmair and Ganzinger's Ordered Resolution Prover." In: *Automated Reasoning*. Edited by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Cham: Springer International Publishing, 2018, pages 89–107. isbn: 978-3-319-94205-6. doi: 10.1007/978-3-319-94205-6_7 (cited on page 42).

[101] Stephan Schulz. "E. A Brainiac Theorem Prover." In: *AI Communications* 15.2–3 (2002), pages 111–126. issn: 0921 -7126. url: https://content.iospress.com/articles/ai-communications/aic260 (cited on pages 5, 12, 25).

[102] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. "Faster, Higher, Stronger. E 2.3." In: *Automated Deduction – CADE 27. Proceedings of the 27th International Conference on Automated Deduction*. CADE 2019 (Natal, Brazil, 27 August 2019–30 August 2019). Edited by Pascal Fontaine. Volume 11716. Lecture Notes in Computer Science. Springer Cham, 21 August 2019, pages 495–507. doi: 10.1007/978-3-030-29436-6_29 (cited on pages 5, 12, 25).

[103] Natarajan Shankar. *Metamathematics, Machines, and Gödel's Proof*. Volume 38. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994 (cited on page 39).

[104] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001 (cited on page 129).

[105] Alexander Steen and Christoph Benzmüller. "The higher-order prover Leo-III." In: *IJCAR 2018*. Edited by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Volume 10900. Lecture Notes in Computer Science. Springer, 2018, pages 108–116. doi: 10.1007/978-3-319-94205-6_8 (cited on page 25).

[106] Christian Sternagel and René Thiemann. "Abstract Rewriting." In: *Archive of Formal Proofs* (2010). https://isa-afp.org/entries/Abstract-Rewriting.html. issn: 2150-914x (cited on page 35).

[107] Christian Sternagel and René Thiemann. "First-Order Terms." In: *Archive of Formal Proofs* (February 2018). https://isa-afp.org/entries/First_Order_Terms.html, Formal proof development. issn: 2150-914x (cited on pages 5, 19, 42).

[108] Lukas Stevens and Tobias Nipkow. "A Verified Decision Procedure for Orders in Isabelle/HOL." In: *Automated Technology for Verification and Analysis. 19th International Symposium*. ATVA 2021 (Gold Coast, Australia, 18 October 2021–22 October 2021). Edited by Zhe Hou and Vijay Ganesh. Volume 12971. Lecture Notes in Computer Science. Springer Cham, 14 October 2021, pages 127–143. isbn: 978-3-030-88884-8. doi: 10.1007/978-3-030-88885-5_9 (cited on pages 3, 12, 31).

[109] Zachary Tatlock and Sorin Lerner. "Bringing Extensibility to Verified Compilers." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Volume 45. 6. 2010, pages 111–121. isbn: 978-1-4503-0019-3. doi: 10.1145/1809028.1806611 (cited on page 129).

[110] René Thiemann and Christian Sternagel. "Certification of termination proofs using CeTA." In: *TPHOLs 2009*. Edited by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Volume 5674. Lecture Notes in Computer Science. Springer, 2009, pages 452–468. doi: 10.1007/978-3-642-03359-9_31 (cited on pages 26, 32).

[111] Sophie Tourret. "A comprehensive framework for saturation theorem proving." In: *Archive of Formal Proofs* (2020). https://www.isa-afp.org/entries/Saturation_Framework.html (cited on page 30).

[112] Sophie Tourret. "The Spawns of the Saturation Framework." In: *7th and 8th Vampire Workshop*. Edited by Laura Kovács and Michael Rawson. Volume 99. EPiC Series in Computing. EasyChair, 2024, pages 1–6. doi: 10.29007/5v9j (cited on page 38).

[113] Sophie Tourret and Jasmin Blanchette. "A Modular Isabelle Framework for Verifying Saturation Provers." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021 (Virtual, Denmark, 17 January 2021–19 January 2021). Edited by Cătălin Hriţcu and Andrei Popescu. Association for Computing Machinery, 20 January 2021, pages 224–237. isbn: 978-1-4503-8299-1. doi: 10.1145/3437992.3439912 (cited on pages 26, 30, 38).

[114] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators." In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, New York, USA: ACM Press, 2008, page 17. isbn: 9781595936899. doi: 10.1145/1328438.1328444 (cited on page 128).

[115] Jean-Baptiste Tristan and Xavier Leroy. "Verified validation of lazy code motion." In: 2009, pages 316–326. doi: http://doi.acm.org/10.1145/1542476.1542512 (cited on page 128).

[116] Jean-Baptiste Tristan and Xavier Leroy. "A Simple, Verified Validator for Software Pipelining (verification pearl)." In: *POPL '10. Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL 2010 (Madrid, Spain, 17 January 2010–23 January 2010). Edited by Jens Palsberg. New York, NY, USA: Association for Computing Machinery, 17 January 2010, pages 83–92. isbn: 978-1-60558-479-9. doi: 10.1145/1706299.1706311 (cited on page 128).

[117] Petar Vukmirović, Jasmin Blanchette, and Stephan Schulz. "Extending a High-Performance Prover to Higher-Order Logic." In: *TACAS 2023, Part II*. Edited by Sriram Sankaranarayanan and Natasha Sharygina. Volume 13994. Lecture Notes in Computer Science. Springer, 2023, pages 111–129. doi: 10.1007/978-3-031-30820-8_10 (cited on pages 12, 25).

[118] Uwe Waldmann. *A Modular Completeness Proof for the Superposition Calculus*. https://nekoka-project.github.io/pubs/isasup_blueprint.pdf. 2024 (cited on pages 26, 27).

[119] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. "A Comprehensive Framework for Saturation Theorem Proving." In: *Journal of Automated Reasoning* 66.4 (November 2022), pages 499–539. issn: 0168-7433. doi: 10.1007/s10817-022-09621-7 (cited on pages 26, 27, 29, 38, 42, 51).

[120] Haichuan Wang, Peng Wu, and David Padua. "Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-Level Specialization." In: *CGO '14. Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2014 (Orlando, FL, USA, 15 February 2014–19 February 2014). Edited by David Kaeli and Tipp Moseley. New York, NY, USA: Association for Computing Machinery, 15 February 2014, pages 295–305. isbn: 978-1-4503-2670-4. doi: 10.1145/2544137.2544153 (cited on pages 7, 101, 130).

[121] Conrad Watt. "Mechanising and Verifying the WebAssembly Specification." In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018 (Los Angeles, CA, USA, 8 January 2018–9 January 2018). Edited by June Andronick and Amy Felty. Association for Computing Machinery, 8 January 2018, pages 53–65. isbn: 978-1-4503-5586-5. doi: 10.1145/3167082 (cited on page 129).

[122] Conrad Watt. "WebAssembly." In: *Archive of Formal Proofs* (April 2018). https://isa-afp.org/entries/WebAssembly.html, Formal proof development. issn: 2150-914x (cited on page 129).

[123] Makarius Wenzel. "Isabelle/Isar. A Generic Framework for Human-Readable Proof Documents." In: *From Insight to Proof. Festschrift in Honour of Andrzej Trybulec*. Edited by Roman Matuszewski and Anna Zalewska. Volume 10(23). Studies in Logic, Grammar, and Rhetoric. University of Białystok, 2007. isbn: 978-83-7431-128-1 (cited on pages 3, 10, 42).

[124] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. "Practical Partial Evaluation for High-Performance Dynamic Language Runtimes." In: *PLDI 2017. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017 (Barcelona, Spain, 18 June 2017–23 June 2017). Edited by Albert Cohen and Martin Vechev. New York, NY, USA: Association for Computing Machinery, 14 June 2017, pages 662–676. isbn: 978-1-4503-4988-8. doi: 10.1145/3062341.3062381 (cited on page 130).

[125] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. "One VM to Rule Them All." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 187–204. isbn: 9781450324724. doi: 10.1145/2509578.2509581 (cited on page 130).

[126] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. "Self-Optimizing AST Interpreters." In: *DLS '12. Proceedings of the 8th symposium on Dynamic languages*. DLS 2012 (Tucson, Arizona, USA, 22 August 2012). Edited by Alessandro Warth. New York, NY, USA: Association for Computing Machinery, 22 August 2012, pages 73–82. isbn: 978-1-4503-1564-7. doi: 10.1145/2384577.2384587 (cited on pages 7, 101, 130).

[127] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers." In: *PLDI '11. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2011 (San Jose, California, USA, 4 June 2011–8 June 2011). Edited by Mary Hall and David Padua. New York, NY, USA: Association for Computing Machinery, 4 June 2011, pages 283–294. isbn: 978-1-4503-0663-8. doi: 10.1145/2345156.1993532 (cited on pages 61, 101).

[128] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations." In: *POPL '12. Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL 2012 (Philadelphia, PA, USA, 25 January 2012–27 January 2012). Edited by Michael Hicks. New York, NY, USA: Association for Computing Machinery, 25 January 2012, pages 427–440. isbn: 978-1-4503-1083-3. doi: 10.1145/2103656.2103709 (cited on page 129).