Finding, Simplifying and Presenting Proofs Automatically

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

eingereicht von

Lydia Kondylidou

31.08.2026

1. Gutachter: Prof. Dr. Jasmin Blanchette
2. Gutachter: Prof. Pascal Fontaine, Ph.D.
3. Gutachter: Prof. Dr. Armin Biere
Tag der mündlichen Prüfung: 02.11.2026

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. 5.)

Hiermit erkläre ich an Eides statt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Kondylidou, Lydia
Name, Vorname

München, 05.11.2026
Ort, Datum

Lydia Kondylidou
Unterschrift Doktorandin

Declaration on Writing Aids

I acknowledge the use of AI-assisted writing tools during the preparation of this thesis. Specifically, I used Claude (Anthropic) and Grammarly throughout this process. These tools were used for correcting grammar and spelling, refining word choice and restructuring sentences; for occasional assistance with LaTeX typesetting; and for reviewing implementation code for correctness and style during the research process. They were not used for generating original research content or contributing to the core scientific findings of the thesis or the included papers.

Zusammenfassung

Automatisierte Theorembeweiser sind Computerprogramme, die formale Beweise für mathematische Theoreme erstellen. Ausgehend von Axiomen und einer zu beweisenden Formel prüfen sie, ob die Formel logisch aus den Axiomen folgt. Ein Beweis in angemessener Zeit ist jedoch nur eine der Herausforderungen; er sollte im Idealfall auch kurz und für Menschen lesbar sein. Diese Arbeit befasst sich mit allen drei Herausforderungen und liefert neue Techniken und Werkzeuge zum automatisierten Finden, Vereinfachen und Darstellen von Beweisen.

Zunächst befassen wir uns mit der Herausforderung, Beweise für Theoreme effizient zu finden. Konkret erweitern wir einen Satisfiability-Modulo-Theories-Solver (SMT-Solver) um neue Fähigkeiten zum Schließen über Quantoren. SMT-Solver bilden eine empirisch erfolgreiche Klasse automatisierter Theorembeweiser, die die Logik erster und zunehmend auch höherer Ordnung unterstützen. Beim Schließen über Quantoren stützen sie sich auf die Quantoreninstanziierung, die universell quantifizierte Variablen auf heuristisch ausgewählte Terme abbildet. Wir stellen MBQI-Enum vor, eine Instanziierungsstrategie, die die bestehende modellbasierte Strategie um syntaxgesteuerte Synthese erweitert. Unsere Strategie zielt auf Domänen ohne etablierte Instanziierungstechniken sowie auf Formeln höherer Ordnung ab, deren universell quantifizierte Variablen auf λ-Terme abgebildet werden müssen. Unser Ansatz unterstützt zudem Hilberts Auswahloperator ε, dessen Behandlung allgemein als schwierig gilt. Der syntaxgesteuerte Enumerator liefert eine breite Palette von Instanziierungskandidaten, wodurch die Erfolgsrate des modernsten SMT-Solvers cvc5 bei Formeln erster und höherer Ordnung erheblich gesteigert wird.

Anschließend beschäftigen wir uns mit der Herausforderung, die von automatisierten Theorembeweisern generierten Beweise zu vereinfachen. Die von ihnen erzeugten Argumentationsketten können sehr lang und schwer nachvollziehbar sein. Im Rahmen des „Equational Theories Project“ bat der Fields-Medaillengewinner Terence Tao die Fachgemeinschaft, Alternativen zu einem komplizierten 62-Schritte-Beweis des Theorembeweisers Vampire zu finden. Wir stellen Krympa vor, ein Werkzeug zur Beweisminimierung, das Brute-Force-Verfahren und Heuristiken kombiniert und dabei Vampire sowie das System Twee nutzt. Krympa reduziert den 62-Schritte-Beweis auf 20 Umschreibungsschritte. In einer empirischen Auswertung von 1.431 Gleichungsproblemen aus demselben Projekt zeigt unser Werkzeug durchweg gute Ergebnisse und reduziert insbesondere einen 151-Schritte-Beweis auf nur 10 Schritte.

Abschließend befassen wir uns mit der Herausforderung, dass die von automatisierten Theorembeweisern generierten Beweise für Menschen in der Regel schwer zu lesen und zu verstehen sind, zum Teil weil es sich fast ausnahmslos um Beweise durch Widerspruch handelt. Bisherige Forschung zur Beweisdarstellung zielt entweder auf die Logik erster Ordnung oder auf rein gleichungsbasiertes Schlussfolgern ab. Wir führen ein strukturiertes Beweisformat für das Horn-Fragment der Logik erster Ordnung mit integrierter Gleichheit ein, das der Twee-Logik entspricht. Das Format stellt direkte Beweise dar, die allgemein als lesbarer gelten als Beweise durch Widerspruch. Es kombiniert gleichungsbasierte Umschreibungsketten mit Ableitungsschritten im menschlichen Stil („have/hence“). Wir stellen zudem einen Übersetzungsalgorithmus vor, der Beweise durch Widerspruch bestehender Theorembeweiser in unser Format überführt.

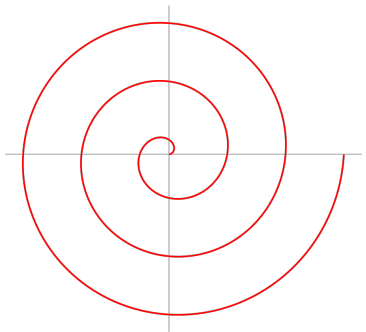
Abstract

Automated theorem provers are computer programs that construct formal proofs of mathematical theorems. Given a set of axioms and a formula to prove, they try to determine whether the formula follows logically from the axioms. Producing a proof in reasonable time, however, is only one of the challenges; the proof should ideally also be concise, and it should be human-readable. This thesis addresses all three challenges, contributing new techniques and tools for finding, simplifying and presenting proofs automatically.

We first address the challenge of efficiently finding proofs for theorems. Specifically, we extend a satisfiability modulo theories (SMT) solver with new quantifier reasoning capabilities. SMT solvers form an empirically successful class of automated theorem provers. They support first-order logic and, increasingly, higher-order logic. To reason about quantifiers in these logics, SMT solvers rely on quantifier instantiation, a class of techniques that map universally quantified variables to heuristically chosen terms. We introduce MBQI-Enum, a quantifier instantiation strategy that extends the existing model-based strategy with syntax-guided synthesis techniques. Our strategy targets domains without well-established quantifier instantiation techniques as well as higher-order formulas whose universally quantified variables need to be mapped to λ-terms. Our approach also supports Hilbert's choice operator ε, which is generally considered to be difficult to reason about. By incorporating a syntax-guided enumerator, the approach generates a broad set of candidate instantiations, substantially increasing the success rate of the state-of-the-art SMT solver cvc5 on both first- and higher-order formulas.

We then address the challenge of simplifying proofs generated by automated theorem provers. Indeed, the chains of argumentation they produce can be very long and difficult to follow. In the context of the Equational Theories Project, Fields medalist Terence Tao asked the community to find alternatives to a complicated 62-step proof found by the Vampire automated theorem prover. We introduce Krympa, a proof minimization tool that combines brute force and heuristics, exploiting both Vampire and another system called Twee. Krympa reduces the 62-step proof to 20 steps, each corresponding to a single rewrite. In an empirical evaluation on 1431 equational problems from the same project, our tool also performs well across the board, reducing in particular a 151-step proof to only 10 steps.

Finally, we address the challenge that proofs generated by automated theorem provers are typically difficult for humans to read and understand, partly because the proofs are almost invariably by contradiction. Existing research on proof presentation either targets first-order logic or focuses on purely equational reasoning. We introduce a structured proof format for the Horn fragment of first-order logic with built-in equality, which corresponds to Twee's logic. The format is designed to express direct proofs, which are generally considered more readable than proofs by contradiction. It combines equational rewriting chains and human-style "have/hence" derivation steps. To interface with existing automated theorem provers, we present a translation algorithm that converts proofs by contradiction to our proof format.



Acknowledgments

First and foremost, I would like to thank my supervisor, Jasmin Blanchette, for his guidance and patience throughout my PhD. He believed in me, and I am more grateful for that than words can say. Jasmin gave me the freedom to explore my own ideas while always being available when I needed direction. His feedback has made every paper substantially better. I remember him reading over my shoulder as I drafted the thesis introduction and asking "Where are the Greeks?" I answered "Which Greeks? The Greek is writing the thesis." He meant the ancient Greek philosophers and mathematicians, whom I had indeed forgotten to include. Jasmin's enthusiasm for the work has been a constant source of motivation. I could not have asked for a better supervisor!

I am grateful to Armin Biere, Pascal Fontaine and Jasmin Blanchette for being on my thesis committee. Armin brings considerable expertise in Boolean satisfiability, which is central to the SMT solving work in this thesis, and I am glad to have his perspective on it. Pascal in particular has been exceptionally engaged throughout. He read all of my papers carefully and offered thoughtful feedback and advice. I also thank Dirk Beyer for chairing my defense and Johannes Kinder for serving as substitute examiner.

I am also deeply grateful to my co-authors Andrew Reynolds, Cesare Tinelli, Marijn Heule, Nicholas Smallbone and Uwe Waldmann. Andrew Reynolds was the first collaborator I worked closely with, at the very start of my PhD. He had enormous patience with me as I was still learning, and he always made time to answer questions or code together. Cesare Tinelli has a remarkable ability to make even the hardest theory clear, and his explanations were invaluable whenever we were stuck. He also took the time to read my thesis carefully and his feedback improved it greatly. Working with Marijn Heule has been a genuine pleasure. We share a competitive drive and a desire to push results as far as they can go, and his energy and enthusiasm made working with him very enjoyable. Uwe Waldmann sat with me on many occasions to work through the harder parts of the formal theory. He knows every detail and has a real gift for explaining them, and working with him is as challenging as it is enjoyable. His feedback on my thesis was thorough and immensely helpful. Finally, Nick Smallbone put in a great deal of work on the Twee extension and was always generous with his time, consistently making time to help me with both our work together and my thesis whenever I needed his advice or assistance.

I would like to thank Thomas Lemberger for providing the thesis template and showing me how to use it.

I also want to thank François Bry. His courses during my bachelor's and master's at LMU are a large part of the reason I decided to do a PhD.

I would also like to thank my group in Munich, whom I will miss enormously. Their support and companionship throughout these years have meant a great deal to me. I also want to thank the group in Iowa, where I spent several months and felt very welcome.

I am grateful to Clark Barrett for offering me a postdoctoral position at Stanford. I cannot wait to start this new chapter.

Finally, and most importantly, I want to thank my family. I left Greece when I was nineteen years old. Saying goodbye at the airport each time I visited and had to leave again was the hardest thing. I cannot begin to express how grateful I am to my parents for standing by me through every decision and every difficulty, for

supporting my dreams, for listening and for understanding my love for research. From a very young age they nurtured my curiosity and always encouraged me to see beyond what I knew. They accepted having me far away, which was its own kind of sacrifice. No matter how far I was or how hard things got, they were always there, a home to return to. They have been proud of this work even when they could not always follow it. I am who I am because of them. I want to give a special mention to my sister, whom I admire and adore more than I can say. She has been with me in Germany for most of my time there, and she has been there for me in more ways than I can count. She is not just my sister but my best friend.

Contents

1 Introduction **1**
1.1 Motivation and Contributions 1
1.2 Structure of This Thesis 3

2 Background **5**
2.1 Equational, First-Order and Higher-Order Logic 5
2.2 Satisfiability Modulo Theories Solving 7
2.3 Resolution Calculus 9
2.4 Superposition Calculus 10

3 Finding Proofs (B.1–B.2) **13**
3.1 Model-Based Quantifier Instantiation 13
3.2 Fast Enumeration of Grammar-Generated Terms 14
3.3 Reasoning about Hilbert's Choice Operator 16
3.4 Related Work 18

4 Simplifying Proofs (B.3) **19**
4.1 Equational Theories 19
4.2 Krympa: An Equational Proof Minimization Tool 20
4.3 Related Work 22

5 Presenting Proofs (B.4) **23**
5.1 Proofs for the Horn Fragment of First-Order Logic 23
5.2 Tälja: A Horn Proof Conversion Tool 24
5.3 Related Work 27

6 Conclusion and Future Directions **29**

Bibliography **31**

A Credits **39**

B Papers **41**
B.1 Augmenting Model-Based Instantiation with Fast Enumeration 42
B.2 Enumerating Choice Terms in Model-Based Quantifier Instantiation . . . 61
B.3 Tao's Equational Proof Challenge Accepted 81
B.4 Readable Proofs for the Horn Fragment 99

Have you tried adding more axioms?

1 Introduction

1.1 Motivation and Contributions

Proofs establish the truth of mathematical statements through rigorous deductive arguments. The quest to automate mathematical reasoning can be traced back to ancient Greece, where Aristotle's systematic study of logical inference and Euclid's axiomatic geometry laid the foundations of formal logic. Centuries later, Leibniz envisioned a *calculus ratiocinator* that could be used to settle any dispute by calculation. This quest gained computational form in 1956 with the Logic Theorist developed by Newell, Shaw and Simon, which proved 38 theorems from Whitehead and Russell's *Principia Mathematica*, and with Robinson's introduction of the resolution principle in 1965, which provided a general inference mechanism for modern theorem proving. Today, computer programs can find proofs for problems too complex for humans to prove by hand. These programs, known as automated theorem provers, take a set of axioms and a formula to prove as input and try to determine whether this formula follows logically from the axioms. However, provers may fail to find a proof at all within reasonable time, and when they do find one, the resulting proof can also be too long to follow or too difficult to understand. These shortcomings point to three fundamental challenges in automated reasoning that this thesis addresses: finding proofs for a broad class of problems efficiently, simplifying the proofs and presenting them in a human-readable format.

Automated theorem provers operate in logical formalisms such as first-order and higher-order logic, which provide precise frameworks for expressing mathematical statements and reasoning about them. First-order logic provides a foundation for set theory and, through it, much of mathematics, while admitting sound and complete proof calculi that make automated proof search feasible in principle. In first-order logic, variables range over individual objects and quantifiers express properties of those objects and relationships between them. Higher-order logic extends first-order logic by allowing quantification over functions and predicates, with λ-abstraction acting as a universal binding mechanism. This additional expressiveness can be used to naturally formalize higher-order functions such as map and fold, set comprehensions, large operators such as $\sum$, $\prod$, $\int$, $\bigcup$ and $\bigcap$, as well as asymptotic notations such as O, Θ and Ω.

Historically, most automated provers were restricted to first-order logic, but they have increasingly been extended to support higher-order logic. An empirically successful class of automated provers is satisfiability modulo theories (SMT) solvers. They combine

a Boolean satisfiability (SAT) solver with decision procedures for interpreted theories such as linear arithmetic, arrays and datatypes. The SAT solver proposes candidate models for the input problem, while the theory solvers check these models for theory-specific conflicts. If a conflict is found, it is added back to the problem and the search continues until either a satisfying assignment or a proof is found. For instance, given the formula $\mathsf{P}(2+2) \wedge \neg\mathsf{P}(4)$, the SAT solver proposes the candidate model in which $\mathsf{P}(2+2)$ is true and $\mathsf{P}(4)$ is false. The theory solvers then detect a conflict: the arithmetic solver evaluates $2+2$ to 4, and congruence closure requires the two atoms to agree. The conflict is added back, no further candidate model exists and the formula is declared unsatisfiable.

Finding proofs efficiently becomes particularly challenging when the input problem contains universal quantifiers ($\forall$) and existential quantifiers ($\exists$). SMT solvers rely on quantifier instantiation strategies, which iteratively map universally quantified variables to ground terms (terms containing no variables), producing instances that are added to the problem. Generating useful instantiations, however, is intricate: in first-order logic, complete strategies exist but may not terminate; in higher-order logic, where instantiations may involve λ-terms, existing strategies offer no completeness guarantees and often give up early. Model-based quantifier instantiation (MBQI), for instance, iteratively refines a candidate model constructed from the quantifier-free part of the problem and creates instantiations using only terms that denote values in that model. As a result, it fails to generate even the identity function $\lambda x.\, x$, which can often lead to a proof. We therefore introduce MBQI-Enum, a new quantifier instantiation strategy that augments MBQI with syntax-guided synthesis (SyGuS) techniques. For each universally quantified variable occurring in the input problem, the strategy constructs a grammar over the problem's symbols and enumerates candidate instantiations guided by that grammar. MBQI-Enum thereby generates a richer set of instantiations, including terms with uninterpreted symbols and λ-abstractions for higher-order formulas (Chapter 3, B.1).

To further improve the effectiveness of MBQI-Enum in higher-order logic, we extend MBQI-Enum to support Hilbert's choice operator ε, which denotes an arbitrary object satisfying a given property, whenever such an object exists. The operator makes it possible to express functions that exist according to the semantics of higher-order logic but cannot otherwise be defined. For instance, if f is a bijection, its inverse can be written as $\lambda y.\, \varepsilon x.\, \mathsf{f}\, x \approx y$. The operator may occur directly in the input problem, but even when it does not, terms containing ε can be the instantiations needed to find a proof. Reasoning about ε is tricky in either case, since it is characterized by a higher-order axiom that quantifies over predicates. By supporting ε in instantiations, MBQI-Enum can find proofs for problems that were previously out of reach (Chapter 3, B.2).

Even when a prover succeeds in finding a proof, the proof may be too long and complex to be useful. During proof search, provers repeatedly apply inference rules, often generating many intermediate steps that make the underlying reasoning difficult to follow. Yet users of these systems are often interested in more than simply knowing that a theorem holds; they also want insight into why it holds. For example, in the context of the Equational Theories Project [15], which studies implications between equational

laws, a 62-step proof found by a state-of-the-art prover was considered too complicated to be instructive. Fields medalist Terence Tao challenged the community to find a simpler proof [66]. Motivated by this challenge, we introduce a proof-minimization approach for proofs in equational logic, the fragment of first-order logic underlying the Equational Theories Project. Given a proof produced by a prover, our approach decomposes it into lemmas, proves each one independently and recombines the resulting subproofs into a shorter proof using heuristics (Chapter 4, B.3).

Even a minimized proof might still be difficult for humans to read. The proof format used by most provers is designed primarily to mimic the low-level operations of the prover rather than to support human understanding. Instead of establishing the formula to prove directly, provers typically work by refutation. They negate the formula to prove, add the negated formula to the axioms and derive a contradiction from the resulting set. To follow a refutational proof, the reader must keep track of which statements depend on the negated formula and mentally reconstruct how the contradiction establishes the original theorem. This task becomes increasingly difficult as proofs grow larger. Moreover, many provers omit details needed to understand how individual proof steps are derived. For example, they often do not indicate the positions at which inferences are applied, leaving readers to reconstruct parts of the argument themselves. As a result, even relatively short proofs can be difficult to read, understand and check. To increase readability of automatically generated proofs, we introduce a structured proof format for the Horn fragment of first-order logic with equality. The format expresses direct proofs by combining equational rewriting chains with "have/hence" derivations, yielding proofs that are arguably more readable than traditional machine-generated proofs. We also present an algorithm that translates refutational proofs into this format and prove the algorithm correct (Chapter 5, B.4).

All contributions are supported by implementations:

- We extended the state-of-the-art SMT solver *cvc5* [11] with our new **MBQI-Enum** strategy (Chapter 3, B.1–B.2).
- We designed and developed **Krympa**, a tool that minimizes equational proofs (Chapter 4, B.3).
- We designed and developed **Tälja**, a tool that translates refutational proofs into our proof format (Chapter 5, B.4).

Our research has been published at the 2025 and 2026 editions of the *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS) and the 2026 edition of the *International Joint Conference on Automated Reasoning* (IJCAR).

1.2 Structure of This Thesis

This thesis connects a selection of my academic publications and is structured as follows. Chapter 1 introduces the motivation and main contributions of this work. Chapter 2 provides the necessary background on the logical foundations and automated reasoning techniques. Chapters 3, 4 and 5 present an overview of the selected publications, covering

my work on finding, simplifying and presenting proofs automatically in the context of first- and higher-order logic. Chapter 6 concludes the thesis and outlines directions for future research. Appendix A details my contributions to each of the papers included in this thesis, whereas Appendix B contains the original printed versions of the TACAS and IJCAR papers and the manuscript of the fourth paper.

2 Background

This chapter introduces the logical formalisms and proof calculi that form the basis for the work presented in the thesis. Section 2.1 covers equational, first-order and higher-order logic. Section 2.2 describes satisfiability modulo theories solving and quantifier instantiation. Section 2.3 presents the resolution calculus and Section 2.4 the superposition calculus.

2.1 Equational, First-Order and Higher-Order Logic

First-order logic (FOL) is a generalization of propositional logic. Propositional logic is a formal system in which statements are represented as propositions, each of which is assigned a truth value. FOL extends this framework by allowing the representation of *individuals*, their properties and relations between them. For example, in propositional logic, the conclusion Q can be derived from the propositions P and $\mathsf{P} \to \mathsf{Q}$. In FOL, the same pattern extends to quantified formulas. Given the fact $\mathsf{P}(\mathsf{a})$ and the universally quantified formula $\forall x.\, \mathsf{P}(x) \to \mathsf{Q}(x)$, the conclusion $\mathsf{Q}(\mathsf{a})$ follows. In this thesis, we work with *many-sorted* FOL, where individuals are partitioned into sets called *sorts*. Each individual belongs to exactly one sort, ensuring that functions and predicates are applied only to arguments of the appropriate sort. Sorts are thus a basic form of *type*.

The language of many-sorted FOL is parameterized by a *signature* $\Sigma = (\mathcal{S}, \Omega, \Pi)$, consisting of a countable set $\mathcal{S}$ of sort symbols, a set Ω of function symbols with sort declarations and a set Π of predicate symbols. Given Σ and a set of variables $\mathcal{X}$, *terms* are built recursively from variables and function applications; a term containing no variables is *ground*. These terms are used to build *atoms*, which are either equations $s \approx t$ between terms, where equality is built into the language rather than part of the signature, or predicate applications $\mathsf{P}(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms. The symbol $\approx$ is conventionally used to distinguish semantic equality from syntactic equality, which is written $=$. Formulas are built from atoms by applying the Boolean connectives $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ and the quantifiers $\forall$ and $\exists$. The *scope* of a quantifier is the subformula to which it applies. Within a formula, a variable occurrence is *bound* if it is within the scope of a quantifier binding it, and *free* otherwise. For example, in $\forall x.\, \mathsf{P}(x, y)$, the occurrence of x in the quantifier's body is bound, and that of y is free. In automated reasoning, it is common to convert formulas to sets of *clauses*, where each clause is an implicitly universally quantified disjunction of literals. A *literal* is an atom or its negation.

The semantics of the language is given by an *interpretation* $\mathcal{A}$, which assigns a non-empty carrier set A to each sort, a function to each function symbol and a relation to each predicate symbol, together with a *variable assignment* $\beta : \mathcal{X} \to A$ mapping each variable to an element of its sort's carrier. A formula is *satisfiable* if some $\mathcal{A}$ and β make it true, *valid* if all do and *unsatisfiable* if none do; an interpretation satisfying a formula is called a *model*.

The syntactic counterpart of a variable assignment is a *substitution* σ, which maps variables to terms of matching sorts. Applying σ to a term t, written $t\sigma$, replaces each variable x in t by $x\sigma$. Two terms s and t are *unifiable* if some substitution σ satisfies $s\sigma = t\sigma$; in that case σ is called a *unifier* of s and t. A unifier σ_1 is *more general* than a unifier σ_2 if there exists a substitution ρ such that $\sigma_2 = \sigma_1 \circ \rho$, i.e., σ_2 is an instance of σ_1. The *most general unifier* $\mathrm{mgu}(s, t)$, when it exists, is a unifier more general than all others. It underlies the inference rules of the resolution and superposition calculi presented in Sections 2.3 and 2.4. FOL is semidecidable, meaning a complete proof system will eventually confirm the validity of any valid formula, but no procedure is guaranteed to terminate on invalid ones. Unification in FOL, however, is decidable [59].

Equational logic is the fragment of many-sorted FOL whose formulas are universally quantified equations between terms of the same sort. Given a set E of equations as axioms, new equations can be derived using five inference rules: *reflexivity*, *symmetry*, *transitivity*, *congruence* (equal subterms yield equal compound terms) and *substitutivity* (replacing equals by equals under any substitution). A finite sequence of such steps constitutes an *equational proof*, and the *equational theory* of E is the set of all equations derivable from it. The semantics is provided by models of E, namely algebras consisting of non-empty carrier sets with interpretations of the function symbols, in which every equation in E holds for all elements of their sorts.

Higher-order logic (HOL) extends FOL by allowing quantification not only over individual variables but also over variables of function and predicate type. In FOL, function and predicate symbols are fixed parts of the signature and cannot be quantified over; HOL removes this restriction. For example, the formula $\forall p.\, \forall x.\, p\, x \to p\, (\mathsf{f}\, x)$ states that the function f preserves every property p.

In this thesis, we work with monomorphic HOL, also known as *simple type theory* [4]. Each term is assigned a type τ defined inductively. The *base sorts* κ retain their role from FOL, the *Boolean type* o classifies truth values and *function types* $\tau_1 \to \tau_2$ classify functions from one type to another. The term language is based on the *simply typed* λ*-calculus*. Given a signature of typed function symbols, terms are built by four rules. We write $t : \tau$ to indicate that term t has type τ. Variables and function symbols declared with type τ are terms of type τ. A λ*-abstraction* $\lambda x^{\tau'}.\, t$, where x has type τ' and body t has type τ, is a term of type $\tau' \to \tau$, with x bound in t; for instance, $\lambda x^{\tau'}.\, x$ has type $\tau' \to \tau'$. An *application* $t\, t'$ of a term $t : \tau' \to \tau$ to a term $t' : \tau'$ is a term of type τ; for instance, if $\mathsf{f} : \tau' \to \tau' \to \tau$ then $\lambda x^{\tau'}.\, \mathsf{f}\, x\, x$ has type $\tau' \to \tau$. A term of type o is called a *formula* and a term of type $\tau_1 \to \cdots \to \tau_n \to o$ is called a *predicate*. Terms are considered equal modulo α*-conversion* $(\lambda x.\, t) = (\lambda y.\, t[x \mapsto y])$, where y does not occur in t, β*-reduction* $(\lambda x.\, t)\, u = t[x \mapsto u]$ and η*-conversion* $(\lambda x.\, t\, x) = t$, where x

does not occur free in t. For example, $(\lambda x.\, \mathsf{f}\, x\, x)\, \mathsf{a}$ β-reduces to $\mathsf{f}\, \mathsf{a}\, \mathsf{a}$, and $\lambda x.\, \mathsf{f}\, x\, x$ can be α-converted to $\lambda z.\, \mathsf{f}\, z\, z$.

The language may further include *Hilbert's choice operator* ε: the term $\varepsilon x^{\tau}.\, \varphi$ denotes some element of type τ satisfying φ if any exists and an arbitrary element of type τ otherwise. Unlike existential quantification, which only asserts the existence of such an element, the choice operator actually yields one. It is characterized by the axiom $\forall y.\, (\exists x.\, y\, x) \rightarrow y\, (\varepsilon x.\, y\, x)$, which states that if some element satisfies y then $\varepsilon x.\, y\, x$ satisfies y as well.

Under *standard semantics*, each function type is interpreted as the set of all functions between the corresponding domains. Although this yields the intended notion of higher-order validity, by Gödel's incompleteness theorem no complete proof system exists for it. Under *Henkin semantics* [28], the domains of function types may be proper subsets of the corresponding full function spaces, provided they satisfy suitable closure conditions. This makes it possible to have sound and complete calculi for HOL. Under this semantics, HOL, like FOL, is semidecidable. In contrast to FOL, unification in HOL is undecidable.

2.2 Satisfiability Modulo Theories Solving

Satisfiability modulo theories (SMT) is the problem of deciding whether a formula is satisfiable with respect to one or more fixed background *theories*. A theory $\mathcal{T}$ consists of a *signature* Σ, collections of sort, function and predicate symbols with fixed arities, together with a class of interpretations specifying the meaning of its symbols and variables. The *constraint satisfiability problem* for $\mathcal{T}$ is the problem of determining, given a conjunction of ground literals, whether it is satisfiable in some interpretation of the theory; a procedure solving this is called a *$\mathcal{T}$-solver*. When the conjunction is $\mathcal{T}$-unsatisfiable, the $\mathcal{T}$-solver also returns a $\mathcal{T}$-unsatisfiable conjunction of the input literals witnessing the inconsistency [12].

Several theories are central to SMT applications. The *theory of equality with uninterpreted functions* has equality as its sole predicate and leaves all function symbols uninterpreted. Its constraint satisfiability problem, for a given set E of ground equalities, is solved by *congruence closure*, which computes from E the smallest equivalence relation on terms closed under the congruence property. In particular, $\mathsf{f}(s_1, \ldots, s_n) \approx \mathsf{f}(t_1, \ldots, t_n)$ holds for any function symbol f and terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ with $s_i \approx t_i$ for all i. A set of equality literals is satisfiable if and only if no disequality holds between two terms in the same congruence class. Consider the set $\{\mathsf{a} \approx \mathsf{b},\ \mathsf{f}(\mathsf{a}) \approx \mathsf{c},\ \mathsf{f}(\mathsf{b}) \not\approx \mathsf{c}\}$. Congruence applied to $\mathsf{a} \approx \mathsf{b}$ yields $\mathsf{f}(\mathsf{a}) \approx \mathsf{f}(\mathsf{b})$; together with $\mathsf{f}(\mathsf{a}) \approx \mathsf{c}$, transitivity gives $\mathsf{f}(\mathsf{b}) \approx \mathsf{c}$, contradicting the disequality. Other widely used theories include linear arithmetic over the integers or reals, bit vectors, datatypes and the theory of arrays, each equipped with dedicated procedures for their constraint satisfiability problem [12]. Multiple theories can be combined modularly, with their respective constraint satisfiability solvers exchanging information via a method such as the Nelson–Oppen method [48].

A widely used architecture for SMT solving is *CDCL(T)* [25], which integrates a SAT solver with one or more $\mathcal{T}$-solvers [12]. Given a quantifier-free formula φ in the

signature of $\mathcal{T}$, the solver constructs a *propositional abstraction* φ' of φ by replacing each atom of φ with a fresh propositional variable. The SAT solver searches for a satisfying Boolean assignment for φ' and, whenever it finds one, constructs a conjunction of $\mathcal{T}$-*literals* and passes it to the $\mathcal{T}$-solver. Each $\mathcal{T}$-literal is an atom of φ if its propositional variable was assigned true, or the negation of that atom if it was assigned false. If this conjunction is $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-satisfiable. Otherwise, the $\mathcal{T}$-solver returns a $\mathcal{T}$-unsatisfiable conjunction of a subset of those literals; the SAT solver learns the disjunction of the negations of the corresponding propositional variables as a new *conflict clause* and backtracks. This process continues until either a $\mathcal{T}$-satisfiable assignment is found or the SAT solver exhausts all Boolean models, establishing $\mathcal{T}$-unsatisfiability.

When φ is quantifier-free, CDCL(T) constitutes a decision procedure for satisfiability in any theory with a decidable constraint satisfiability problem. When φ contains quantifiers, however, CDCL(T) does not directly apply, since the $\mathcal{T}$-satisfiability of quantified formulas is generally undecidable. Modern SMT solvers address this gap as follows. The formula φ may contain quantified subformulas at any level, whether top-level or nested. Quantified formulas are treated as propositional variables in the abstraction of φ. To reason about the quantified formulas, the SMT solver converts them to ground ones. Existential quantifiers are eliminated by *Skolemization*: in a formula $\forall x_1, \ldots, x_n. \exists y. \psi$, the variable y is removed and all its free occurrences in ψ are replaced by a fresh function symbol $\mathsf{f}(x_1, \ldots, x_n)$ of arity n, yielding a purely universal formula. Each universally quantified formula $\forall x_1, \ldots, x_n. \psi$ is then conjoined with a number of its instances, obtained from ψ by substituting ground terms for the bound variables. The resulting ground instances are quantifier-free and can be processed by CDCL(T) [12]. For example, when $\forall x. \psi$ appears positively, as in $\varphi \vee \forall x. \psi$, it is abstracted as $\varphi \vee a$ and the solver adds $a \to \psi[x \mapsto t]$ for a ground term t. When it appears negated, as in $\varphi \wedge \neg\forall x. \psi$, it is abstracted as $\varphi \wedge \neg a$, and the solver Skolemizes $\neg\forall x. \psi$ (equivalent to $\exists x. \neg\psi$) by introducing a fresh constant sk and adding $\neg a \to \neg\psi[x \mapsto \mathsf{sk}]$.

To generate ground instances of universally quantified formulas, SMT solvers rely on *instantiation strategies* [12]. An instantiation strategy computes, for each universally quantified formula, a set of substitutions mapping the bound variables to ground terms. The most widely used strategy is *E-matching* [45], a heuristic and typically incomplete technique that chooses substitutions by matching ground terms with patterns, known as *triggers* [45]. Given a formula $\forall x. \psi$, the SMT solver identifies a subterm $t[x]$ of ψ in which x occurs as a trigger. Two terms are $\mathcal{T}$*-equivalent* if they are equal in all models of $\mathcal{T}$. It then searches for a ground term g and a subterm s of g such that $t[s]$ is $\mathcal{T}$-equivalent to g, and generates the instance $\psi[x \mapsto s]$ [12]. Another widely used strategy is *model-based quantifier instantiation* (MBQI) [26], which uses a candidate model to guide instance generation; we discuss it in detail in the next chapter. For first-order logic with theories, some strategies are refutationally complete. The SMT solver is then a semidecision procedure [12].

2.3 Resolution Calculus

The resolution calculus is a refutational calculus for FOL. Proving a conjecture φ from axioms Γ proceeds by clausifying $\Gamma \cup \{\neg\varphi\}$ into an equisatisfiable set of clauses F and searching for a derivation of the empty clause $\bot$, which witnesses the unsatisfiability of F and hence the validity of φ with respect to Γ. A clause is a finite multiset of literals, written as a disjunction $l_1 \vee \cdots \vee l_n$. Clauses may contain variables, which are implicitly universally quantified; for instance, the clause $\mathsf{P}(x) \vee \mathsf{Q}(y)$ is equivalent to the formula $\forall x, y.\, \mathsf{P}(x) \vee \mathsf{Q}(y)$. Since clauses are multisets, the order of literals does not matter, so $\mathsf{P}(x) \vee \mathsf{Q}(y)$ and $\mathsf{Q}(y) \vee \mathsf{P}(x)$ denote the same clause. The empty clause, denoted $\bot$, is the empty multiset. Clausification [53] converts the input formula into the initial set of clauses F. New clauses are derived from existing ones via *inference rules*. Each rule takes one or more *premise* clauses and produces a *conclusion* clause. During proof search, resolution provers repeatedly apply inferences on the currently derived set of clauses and derive new conclusions, which are added to the set. The search continues until $\bot$ is derived or no more inferences can be drawn [8]. The resolution calculus is sound and, if inferences are performed fairly (i.e., no applicable inferences are postponed forever), it is also complete.

The core inference rules of the calculus are *resolution* and *factoring*. Resolution takes two clauses $C \vee A$ and $\neg A' \vee D$, where A and A' are unifiable atoms, and derives $C \vee D$ under their most general unifier. Factoring merges two unifiable positive atoms in a clause:

$$\frac{C \vee A \qquad \neg A' \vee D}{(C \vee D)\sigma}\ \textsc{Res} \qquad \frac{C \vee A \vee A'}{(C \vee A)\sigma}\ \textsc{Fact}$$

Both rules carry the side condition $\sigma = \mathrm{mgu}(A, A')$, so the conclusion simultaneously represents all ground conclusions derivable from ground instances of the premises. A schematic premise such as $\neg A' \vee D$ stands for an arbitrary clause containing the literal $\neg A'$ anywhere and zero or more additional literals forming D. Variables in the premises of the resolution rule are implicitly renamed apart before the rule is applied. Soundness of resolution is immediate by a case analysis on the truth value of A. The subclauses C and D may be empty. In particular, in the Res rule, if both C and D are $\bot$, the conclusion is also $\bot$.

To restrict the search space, a fixed *ordering* $>$ on ground atoms can be introduced. It is extended to literals by treating a negative literal as larger than the corresponding positive one, and to clauses by comparing them as multisets of literals [8]. The resulting calculus, *ordered resolution*, applies restrictions on which literals may be resolved, based on the ordering. This calculus is also complete.

Proof search can generate a large number of clauses. *Redundancy* criteria are used to delete clauses that are not necessary for finding $\bot$. A ground clause C is *redundant* with respect to a set of ground clauses F if C is entailed by clauses in F that are strictly smaller with respect to $>$. A general clause C is redundant with respect to a set of clauses F if every ground instance of C is redundant with respect to the ground instances of F. The most common form of redundancy is *subsumption*. A clause C subsumes D if some

substitution σ satisfies $C\sigma \subseteq D$ as multisets of literals. A clause C properly subsumes D if C subsumes D but D does not subsume C. In this case D can be safely deleted. For example, $\mathsf{P}(x)$ properly subsumes $\mathsf{P}(x) \vee \mathsf{Q}(y)$.

> **Example 1.** Consider refuting $\{\neg\mathsf{P}(x) \vee \mathsf{Q}(x),\ \mathsf{P}(\mathsf{a}),\ \neg\mathsf{Q}(\mathsf{a})\}$. Applying Res to the first two clauses with $\sigma = \{x \mapsto \mathsf{a}\}$ gives $\mathsf{Q}(\mathsf{a})$; a further Res step with $\neg\mathsf{Q}(\mathsf{a})$ then derives $\bot$.

2.4 Superposition Calculus

The superposition calculus extends resolution with native support for equality, avoiding the combinatorial explosion that arises from axiomatizing equality with reflexivity, symmetry, transitivity and congruence axioms. Instead, equality is treated as *built in*. Since any predicate atom $\mathsf{P}(t_1, \ldots, t_n)$ can be encoded as $\mathsf{P}(t_1, \ldots, t_n) \approx \mathsf{true}$, we may assume for simplicity that the signature contains no predicate symbols except built-in equality. A *clause* is a multiset of literals $s \approx t$ or $s \not\approx t$, where $s \approx t$ and $t \approx s$ denote the same literal, as do $s \not\approx t$ and $t \not\approx s$. An *interpretation* (D, J) assigns to each function symbol f a function $\mathsf{f}^J : D^n \to D$ over a non-empty domain D; equality is interpreted as identity. The superposition calculus is sound and, if inferences are performed fairly, also complete [9].

The fundamental inference rule for equality reasoning is *paramodulation*. In the ground case, it rewrites a designated occurrence of t in a literal $L[t]$ using an equation $t \approx t'$.

$$\frac{C \vee t \approx t' \qquad D \vee L[t]}{C \vee D \vee L[t']}\ \textsc{Para}$$

The superposition calculus adds an *ordering* $>$ on terms to paramodulation, imposing ordering restrictions on which literals and sides may be used for rewriting. These restrictions prune the search space substantially while preserving completeness.

The nonground calculus consists of three rules. The main rule, Sup, extends Para to the nonground case by unifying the rewriting term t with a subterm u of the right premise via a most general unifier. *Equality resolution* (ERes) removes a disequality literal $s \not\approx t$ by unifying s and t, since the resulting instantiated literal is trivially false. *Equality factoring* (EFact) merges two positive equality literals with unifiable left-hand sides, similarly to the Fact rule in resolution.

$$\frac{C \vee t \approx t' \qquad D \vee L[u]}{(C \vee D \vee L[t'])\sigma}\ \textsc{Sup} \qquad \frac{C \vee s \not\approx t}{C\sigma}\ \textsc{ERes} \qquad \frac{C \vee s \approx t \vee s' \approx t'}{(C \vee t \not\approx t' \vee s \approx t')\sigma}\ \textsc{EFact}$$

The substitution σ is $\mathrm{mgu}(t, u)$ in Sup, $\mathrm{mgu}(s, t)$ in ERes and $\mathrm{mgu}(s, s')$ in EFact. In Sup, the subterm u is additionally required not to be a variable; this condition is not needed for soundness or completeness but is crucial for efficiency.

The redundancy criterion of Section 2.3 can be extended to support equality reasoning. Using entailment with respect to equality in place of plain entailment, a clause

C is redundant with respect to a set of equational clauses F if each ground instance of C is redundant with respect to the ground instances of F. This makes it possible, for example, to delete $\mathsf{f}(x) \approx \mathsf{h}(x)$ when $\mathsf{f}(x) \approx \mathsf{g}(x)$ and $\mathsf{g}(x) \approx \mathsf{h}(x)$ are both in F and $\mathsf{g}(x)$ is small enough with respect to $>$, or to discard tautologies of the form $C \vee t \approx t$. *Demodulation*, replacing an occurrence of a larger side t in a literal by a smaller side t' using an equation $t \approx t' \in F$, is another key simplification that removes clauses made redundant by their neighbors.

Example 2. Consider refuting $\{\mathsf{b} \not\approx \mathsf{a},\ x \approx \mathsf{a} \vee \mathsf{f}(x) \approx \mathsf{g}(x),\ \mathsf{f}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{b})\}$. Applying Sup to the second and third clause with $\sigma = \{x \mapsto \mathsf{b}\}$ yields $\mathsf{b} \approx \mathsf{a} \vee \mathsf{g}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{b})$. ERes removes the disequality literal, giving $\mathsf{b} \approx \mathsf{a}$. A further Sup step on $\mathsf{b} \approx \mathsf{a}$ and the first clause produces $\mathsf{a} \not\approx \mathsf{a}$, from which ERes derives $\bot$.

3 Finding Proofs (B.1–B.2)

This chapter addresses the first challenge described in Chapter 1: finding proofs efficiently in the presence of quantifiers. As discussed there, existing strategies fail on problems in uninterpreted theories and higher-order logic. Section 3.1 recalls model-based quantifier instantiation (MBQI) and its limitations. Section 3.2 introduces MBQI-Enum, which overcomes these limitations by augmenting MBQI with a syntax-guided enumerator. Section 3.3 further extends MBQI-Enum to generate instantiations involving Hilbert's choice operator ε.

3.1 Model-Based Quantifier Instantiation

In an SMT solver, the input problem may contain both ground and quantified formulas. During solving, CDCL(T) processes the current conjunction of ground formulas (formulas built from ground terms), which starts from the ground formulas of the input and grows as quantifier instantiation strategies generate instances of the universally quantified formulas, as described in Chapter 2. Model-based quantifier instantiation (MBQI) [26] is a quantifier instantiation strategy developed for first-order logic and later applied to higher-order logic as well. It constructs a candidate model M satisfying the current ground conjunction, and uses it to identify substitutions that falsify the quantified formulas. We write s^M for the value assigned to a symbol or term s by M. For each universally quantified formula $q = \forall x_1, \ldots, x_n. \psi$, MBQI evaluates the negated body $\neg\psi$ under M. If $\neg\psi^M$ is $\mathcal{T}$-satisfiable, a witness model M' exists and MBQI returns the substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where each t_i is the value of x_i in M'. The instance $\psi\sigma$ is appended to the ground problem and the process repeats with a new candidate model for the augmented problem. Each new instance further constrains the model. The process terminates when the model satisfies all quantified formulas, witnessing satisfiability, or when the ground problem becomes unsatisfiable.

Example 3. Let $\mathsf{f} : Int \to Int$ and $\mathsf{a} : Int$. Let φ be the unsatisfiable formula

$$(\forall x. \mathsf{f}\, x > x) \wedge \mathsf{f}\, \mathsf{a} \leq \mathsf{a}$$

where $x : Int$. MBQI constructs a candidate model M satisfying the ground part $\mathsf{f}\, \mathsf{a} \leq \mathsf{a}$, with $\mathsf{a}^M = 0$ and $\mathsf{f}^M = \lambda x. 0$. Evaluating $\neg(\mathsf{f}\, x > x)$ under M gives $0 \leq x$,

which is $\mathcal{T}$-satisfiable, for example at $x = 0 = \mathsf{a}^M$. MBQI returns $\{x \mapsto \mathsf{a}\}$ and adds the instance $\mathsf{f}\,\mathsf{a} > \mathsf{a}$, which contradicts $\mathsf{f}\,\mathsf{a} \leq \mathsf{a}$ in the ground problem.

MBQI has two limitations that cause it to either give up early or run forever without finding a proof. The first concerns the source of instantiation terms. Since M is a theory model, it interprets every symbol over a domain of concrete values; the value of an application $\mathsf{f}(t_1, \ldots, t_n)$ in M is some concrete domain element, not a term. Substitutions returned by MBQI therefore consist entirely of theory values (concrete elements of the theory domain) and never include terms built from the problem's own *uninterpreted* function symbols, that is, symbols not assigned a fixed meaning by the background theory, as opposed to theory symbols such as $+$ or $\leq$. When the desired instantiation requires such a term, MBQI cycles through theory values infinitely without finding it. The second limitation concerns higher-order variables. For a variable y of function type $\tau_1 \to \tau_2$, MBQI can only return a value of that type as it appears in the model, which is a concrete function over the model domain rather than a λ-term. Terms such as the identity $\lambda x.\, x$ or any λ-abstraction built from the problem's own symbols lie outside MBQI's reach. Consider the unsatisfiable formula $(\forall y.\, \neg \mathsf{p}\,(y\,\mathsf{a})) \wedge \mathsf{p}\,\mathsf{a}$, where $y : Int \to Int$. MBQI generates substitutions such as $\{y \mapsto \lambda x.\, 0\}$, $\{y \mapsto \lambda x.\, 1\}$, …, and the solver does not terminate. The useful instantiation $\{y \mapsto \lambda x.\, x\}$, however, cannot be found because $\lambda x.\, x$ is not a value in any model of the quantifier-free part. Together, these two limitations cause MBQI to fail on problems in uninterpreted first-order theories and on higher-order problems.

3.2 Fast Enumeration of Grammar-Generated Terms

We lift both limitations by introducing MBQI-Enum, a new quantifier instantiation strategy. Syntax-guided synthesis (SyGuS) [56] is a grammar-based approach to term enumeration. A SyGuS enumerator systematically generates all terms derivable from a context-free grammar, in order of increasing size. MBQI-Enum integrates a SyGuS enumerator into MBQI to produce a richer set of candidate instantiations.

MBQI-Enum first invokes MBQI to obtain an initial substitution σ. If σ is empty, the formula is already satisfied by the current model and the strategy returns immediately. Otherwise, it generates additional substitutions by extending σ using the SyGuS enumerator. Let $q = \forall y_1, \ldots, y_n.\, \psi$ be the quantified formula being instantiated. For each variable y_i, the strategy first determines a symbol set S of uninterpreted symbols and variables from the formula. Three Boolean options control which symbols are included: the *ext_vars* option adds bound variables in q yet to be instantiated; the *syms_local* option adds uninterpreted function symbols from q; the *syms_global* option adds all uninterpreted function symbols from the entire input formula. Using S, the strategy then constructs a SyGuS grammar $G_i = (g_0, \mathcal{G}_i, R_i)$, where g_0 is the initial symbol and $\mathcal{G}_i$ is a set of nonterminal symbols with $g_0 \in \mathcal{G}_i$. The production rules R_i have the form $g ::= t$, where $g \in \mathcal{G}_i$ and t is a term built from symbols in S, nonterminals in $\mathcal{G}_i$ and theory symbols from $\mathcal{T}$. For higher-order variables, g_0 expands to a λ-abstraction $\lambda x_1, \ldots, x_k.\, g'$, where $g' \in \mathcal{G}_i$ is a nonterminal generating the body and $x_1, \ldots, x_k$ are

the bound variables of y_i; in this case, t may also contain $x_1, \ldots, x_k$. For example, for a higher-order variable y_i of arity k, the grammar includes rules such as

$$g_0 ::= \lambda x_1, \ldots, x_k.\, g' \qquad g' ::= x_1 \mid \cdots \mid x_k$$

The grammar generates terms of the same type as the variable y_i. MBQI-Enum then enumerates candidate terms from the grammar. For each candidate term e, the strategy creates a new substitution $\sigma' = \sigma[y_i \mapsto e]$, obtained from σ by mapping y_i to e, and checks whether the negated body $\neg\psi$ of q, after applying σ', is $\mathcal{T}$-satisfiable. This check acts as a semantic filter: if the negated body under the updated substitution is still satisfiable, the term refutes the current model and σ is updated to σ'. If no enumerated term passes the check, MBQI-Enum falls back to the original MBQI substitution for y_i.

Returning to the example $(\forall y.\, \neg \mathsf{p}\,(y\,\mathsf{a})) \wedge \mathsf{p}\,\mathsf{a}$ above, recall that MBQI fails because it cannot generate the instantiation term $\lambda x.\, x$ for y, since it is not a value in any theory model. By contrast, MBQI-Enum first builds the following grammar for y:

$$g_0 ::= \lambda x.\, g_1 \qquad g_1 ::= x \mid 0 \mid 1 \mid g_1 + g_1 \mid g_1 - g_1$$

The strategy then constructs $\lambda x.\, x$ from the grammar and enumerates it. Instantiating y with $\lambda x.\, x$ and β-reducing gives $\neg \mathsf{p}\,\mathsf{a}$, which is conjoined with the universally quantified formula; the SMT solver finds the result to be $\mathcal{T}$-unsatisfiable.

Several factors contribute to MBQI-Enum's efficiency. The SyGuS enumerator generates terms in order of increasing size, so simpler terms are considered first. Since the grammar is built from the formula's own uninterpreted symbols, the terms it generates are directly relevant to the problem, and the correct instantiation tends to appear early in the enumeration. Once a suitable term is found, enumeration stops immediately. Moreover, the grammar for each variable is constructed once and remains fixed throughout solving; the enumerator caches its state between iterations, so subsequent calls resume from where the previous one left off. Terms skipped in an earlier iteration because they did not yet refute the model may become useful in later iterations as new ground formulas are added.

Example 4. Let $\mathsf{f} : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$, $\mathsf{suc} : \mathit{Int} \to \mathit{Int}$, $\mathsf{p} : \mathit{Int} \to o$ and $\mathsf{a}, \mathsf{b} : \mathit{Int}$. Let φ be the unsatisfiable formula

$$(\forall y.\, \mathsf{p}\,(y\,\mathsf{a}\,\mathsf{b})) \wedge \neg \mathsf{p}\,(\mathsf{f}\,\mathsf{a}\,(\mathsf{suc}\,\mathsf{b}))$$

where $y : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$. MBQI generates substitutions $\{y \mapsto \lambda x, z.\, t\}$ for ground values t and does not terminate, since any function of the form $\lambda x, z.\, t$ is constant and cannot equal the required $\lambda x, z.\, \mathsf{f}\,x\,(\mathsf{suc}\,z)$. MBQI-Enum constructs the following grammar for y using the symbol set $S = \{\mathsf{f}, \mathsf{suc}, \mathsf{a}, \mathsf{b}\}$:

$$\begin{aligned} g_0 &::= \lambda x, z.\, g_1 \\ g_1 &::= x \mid z \mid \mathsf{f}\,g_1\,g_1 \mid \mathsf{suc}\,g_1 \mid \mathsf{a} \mid \mathsf{b} \end{aligned}$$

> The strategy enumerates terms generated from the grammar in order of increasing size, starting from the smallest terms such as $\lambda x, z.\, x$ and $\lambda x, z.\, z$. The target term $\lambda x, z.\, \mathsf{f}\, x\, (\mathsf{suc}\, z)$ has total size 4. When the enumerator reaches it, the strategy forms the substitution $\{y \mapsto \lambda x, z.\, \mathsf{f}\, x\, (\mathsf{suc}\, z)\}$. Instantiating y with $\lambda x, z.\, \mathsf{f}\, x\, (\mathsf{suc}\, z)$ and β-reducing gives $\mathsf{p}\, (\mathsf{f}\, \mathsf{a}\, (\mathsf{suc}\, \mathsf{b}))$. The universally quantified formula is then conjoined with this instance. Now, the SMT solver finds φ to be $\mathcal{T}$-unsatisfiable.

The evaluation uses the default MBQI-Enum configuration, in which the options *syms_local*, *syms_global* and *ext_vars* are all enabled, and a timeout of 60 seconds per benchmark. On 21,605 first-order SMT-LIB benchmarks, MBQI-Enum is evaluated against other cvc5 [11] quantifier instantiation strategies, namely enumerative instantiation, syntax-guided quantifier instantiation, counterexample-guided instantiation and MBQI, as well as the state-of-the-art SMT solvers Bitwuzla [49], Boolector [51] and Z3 [46]. MBQI-Enum solves 15,613 problems in total (2755 satisfiable and 12,858 unsatisfiable), more than any other system. It solves 621 more problems than MBQI and 597 more than Z3, the nearest overall competitor. On 2762 higher-order TPTP TH0 benchmarks, MBQI-Enum is also compared with the state-of-the-art higher-order automated theorem provers Vampire [38] and Zipperposition [13]. MBQI-Enum solves 1799 problems in total (129 satisfiable and 1670 unsatisfiable), compared with 1758 for MBQI and 1382 for syntax-guided quantifier instantiation. It outperforms Vampire (1763) and Zipperposition (1499), and notably solves 129 satisfiable problems, compared with 6 for Vampire and none for Zipperposition.

3.3 Reasoning about Hilbert's Choice Operator

We extend MBQI-Enum to also generate instantiations involving Hilbert's choice operator ε. As described in Chapter 2, the term $\varepsilon x^{\tau}.\, \varphi$ denotes some element of type τ satisfying φ if any exists. Even when ε is absent from the input problem, choice terms can be the instantiations needed to find a proof. For instance, if f is injective, its left inverse can be expressed as $\lambda y.\, \varepsilon x.\, \mathsf{f}\, x \approx y$, a term that cannot in general be written without ε.

The extended strategy augments the MBQI-Enum grammar for each variable y_i to include ε terms. The base grammar (Section 3.2) uses a set S of uninterpreted symbols and has an initial nonterminal g_0 that expands to a λ-abstraction whose body is generated by g_1, the nonterminal for terms of the return type τ of y_i. A fresh variable $x : \tau$ is introduced. A predicate grammar is constructed over the extended symbol set $S \cup \{x\}$, with rules

$$p_0 ::= \mathsf{true} \mid \mathsf{false} \mid p_1 \approx p_1 \mid \neg p_0 \mid p_0 \wedge p_0$$
$$p_1 ::= x_1 \mid \cdots \mid x_k \mid x$$

where $x_1, \ldots, x_k$ are the variables bound in y_i but appearing free in p_0. The base grammar is then augmented with the new production $g_1 ::= \varepsilon x.\, p_0$, yielding

$$g_0 ::= \lambda x_1, \ldots, x_k.\, g_1$$
$$g_1 ::= \varepsilon x.\, p_0 \mid x_1 \mid \cdots \mid x_k$$

Since Hilbert's choice operator is not directly supported by cvc5, each subterm $\varepsilon x.\, \varphi(\bar{z}, x)$ in an enumerated term is abstracted into a fresh Skolem symbol h applied to the free variables $\bar{z}$, converting the choice term into a form the solver can reason about directly. Since h is a fresh symbol about which the solver knows nothing, the strategy adds a characteristic lemma

$$\forall \bar{z}.\, \neg(\exists x.\, \varphi(\bar{z}, x)) \vee \varphi(\bar{z},\, \mathsf{h}\, \bar{z})$$

which states that $\mathsf{h}\, \bar{z}$ satisfies φ whenever any element of its type does, mirroring the semantics of the original ε term. The characteristic lemma and the resulting instance of the quantified formula are then added to the input problem.

Example 5. Let $\mathsf{f} : u \to u$ be an uninterpreted function and let φ be the unsatisfiable formula

$$(\forall x, y.\, \mathsf{f}\, x \approx \mathsf{f}\, y \to x \approx y) \;\wedge\; (\forall k.\, \exists z.\, k\, (\mathsf{f}\, z) \not\approx z)$$

where $x, y, z : u$ and $k : u \to u$. The first conjunct states that f is injective; the second denies the existence of a left inverse. Without the ε extension, MBQI-Enum constructs a grammar for k that generates only ε-free terms. The strategy enumerates $\lambda y.\, y$, $\lambda y.\, \mathsf{f}\, y$, $\lambda y.\, \mathsf{f}\, (\mathsf{f}\, y)$, and so on, but none of these can serve as a left inverse of an injective function, so the solver does not terminate. With the extension and the symbol set $S = \{\mathsf{f}\}$, the grammar for k, with initial nonterminal g_0, becomes

$$\begin{aligned} g_0 &::= \lambda y.\, g_1 \\ g_1 &::= \varepsilon x.\, p_0 \mid y \mid \mathsf{f}\, g_1 \\ p_0 &::= \mathsf{true} \mid \mathsf{false} \mid p_1 \approx p_1 \mid \neg p_0 \mid p_0 \wedge p_0 \\ p_1 &::= y \mid x \mid \mathsf{f}\, p_1 \end{aligned}$$

The strategy enumerates the term $\lambda y.\, \varepsilon x.\, \mathsf{f}\, x \approx y$, which denotes the left inverse of f. The ε subterm is replaced by a fresh Skolem symbol $\mathsf{h} : u \to u$, and the characteristic lemma

$$\forall y.\, \neg(\exists x.\, \mathsf{f}\, x \approx y) \vee \mathsf{f}\, (\mathsf{h}\, y) \approx y$$

is conjoined with φ. Applying the substitution $\{k \mapsto \lambda y.\, \mathsf{h}\, y\}$ and β-reducing gives $\exists z.\, \mathsf{h}\, (\mathsf{f}\, z) \not\approx z$, which is conjoined with φ. Skolemizing the existential introduces a fresh constant sk, yielding the ground fact $\mathsf{h}\, (\mathsf{f}\, \mathsf{sk}) \not\approx \mathsf{sk}$. E-matching then instantiates the characteristic lemma and the injectivity axiom with appropriate terms, and the quantifier-free subsolver finds φ to be $\mathcal{T}$-unsatisfiable.

The extended strategy is evaluated on 3757 higher-order TPTP TH0 benchmarks. It solves 2382 problems in total (204 satisfiable and 2178 unsatisfiable), compared with 2359 for MBQI-Enum. It outperforms Satallax [16], Vampire [38] and Zipperposition [13]; the nearest competitor, Satallax, solves 196 satisfiable and 2162 unsatisfiable problems.

3.4 Related Work

Besides MBQI-Enum, other quantifier instantiation strategies work on higher-order problems. E-matching [45] instantiates quantified variables by matching patterns against the ground terms available in the current context. It is the most widely deployed strategy in SMT solvers. Higher-order MBQI [26] constructs candidate models providing interpretations for functions, enabling ground instantiations of quantified formulas. SyQI [50] further improves instantiation by selecting candidate terms using syntactic templates and heuristics derived from the problem's structure, guided by a fixed grammar. Conflict-based instantiation [58] searches for a substitution that directly induces a ground conflict, while enumerative instantiation [55] systematically explores all current ground terms. Counterexample-guided instantiation [57] is refutationally complete for specific first-order theories. These strategies are complementary and all available in cvc5. None of them provides special support for Hilbert's choice.

Recent work has also explored the combination of instantiation strategies. Dančo et al. [19] investigate the formal relation between MBQI and enumerative instantiation and propose a hybrid algorithm that draws on both. Jakubův and Janota [32] derive a probabilistic context-free grammar from instantiations observed during solving and use it to guide the generation of candidate terms.

Other higher-order provers support Hilbert's choice in various ways. Leo-III [63] integrates ε into its extensional higher-order paramodulation calculus and supports Henkin semantics with choice. Satallax [16] implements specialized tableau inference rules that directly reason about ε in a sound and refutationally complete way [10]. Vampire [14] supports Hilbert's choice either through a Leo-III-style inference or by introducing the choice axiom. Zipperposition [13] includes a dedicated choice axiom within its higher-order superposition calculus, allowing it to reason about ε terms as needed.

4 Simplifying Proofs (B.3)

This chapter addresses the second challenge described in Chapter 1: minimizing equational proofs found by automated theorem provers. Section 4.1 presents equational theories and the context motivating proof minimization. Section 4.2 introduces Krympa, a tool we developed to minimize equational proofs.

4.1 Equational Theories

As described in Chapter 2, equational logic is the fragment of first-order logic whose formulas are universally quantified equations. The superposition calculus [9] provides a refutationally complete procedure for equational logic. In practice, however, the proofs it produces can be overly complicated.

The Equational Theories Project [15], launched by Fields medalist Terence Tao in September 2024, studies implications between equational laws for magmas. A *magma* is a basic algebraic structure consisting of a set equipped with a single binary operation $\diamond$. The project's first phase, concluded in April 2025, focused on laws containing at most four applications of $\diamond$. The problems considered in this phase all fall within equational logic, where each problem consists of a single universally quantified equation as the axiom and a universally quantified equation as the conjecture. The project uses the Lean proof assistant to formalize the results. When an implication holds, a proof is formalized in Lean; when it does not, a finite counterexample model is provided in Lean. Many implications are established by human contributors, but for those too difficult to prove by hand, the project relies on the automated prover Vampire [38]. The proofs Vampire produces, however, are often very long and difficult to read or reuse.

One of the most difficult implications in the project is $650 \Rightarrow 448$, where axiom 650 is $\forall x, y, z.\, x \approx x \diamond (y \diamond ((z \diamond x) \diamond y))$ and conjecture 448 is $\forall x, y, z.\, x \approx x \diamond (y \diamond (z \diamond (x \diamond z)))$. Vampire found a 62-step superposition proof of this implication (excluding clausification and Skolemization). This proof was too long to follow, which led Terence Tao to challenge the community to find "an alternate proof, by whatever means you wish—human, semi-automated, or automated" [66].

4.2 Krympa: An Equational Proof Minimization Tool

We address Tao's challenge by introducing Krympa, a tool that takes an equational proof generated by Vampire as input and produces a shorter, more readable proof of the same theorem. Given a refutational proof of a conjecture C from axiom A, Krympa converts it into a direct proof, extracts intermediate lemmas, proves each lemma independently using two provers, combines the resulting subproofs into a shorter proof using heuristics and outputs a Lean [47] formalization.

Vampire [38] is a saturation-based prover based on the superposition calculus. It works by refutation; it establishes the conjecture C from the axioms A by negating the conjecture and deriving a contradiction. Since this proof is refutational rather than a direct derivation of C from A, it cannot be decomposed into reusable lemmas; additionally, Skolem constants introduced for the existential witnesses of $\neg C$ cannot appear in direct equational claims. Krympa therefore begins by converting the refutational proof into a direct proof that derives C from A as a sequence of lemmas. The conversion first introduces existential quantifiers for Skolem constants and universal quantifiers for variables. It then takes the contrapositive of superposition inferences in which a premise and the conclusion are both disequalities, turning the disequality conclusion into a positive equality. For example, the inference

$$\frac{\mathsf{h}(\mathsf{a}, y) \approx \mathsf{b} \qquad \mathsf{h}(x, \mathsf{sk}) \not\approx x}{\mathsf{b} \not\approx \mathsf{a}}$$

becomes, after introducing quantifiers and taking the contrapositive,

$$\frac{\forall y.\ \mathsf{h}(\mathsf{a}, y) \approx \mathsf{b} \qquad \mathsf{b} \approx \mathsf{a}}{\forall z.\ \exists x.\ \mathsf{h}(x, z) \approx x}$$

Equality resolution inferences from a premise $t \not\approx t$ are omitted, since their contrapositives derive tautologies $t \approx t$.

Krympa uses the provers Twee and Vampire during proof minimization. Twee [61] is an automated theorem prover specialized for equational reasoning. It is based on the unfailing completion procedure [7]. Twee's proofs are structured as a sequence of lemmas, where the lemmas and the conjecture are proved by chains of equalities. Since Twee's proofs are already direct, they require no conversion.

Having converted the refutational proof into a direct proof expressed as a sequence of lemmas, referred to as the *baseline proof*, Krympa proceeds to prove each lemma independently using Twee and Vampire. For each lemma ℓ, Krympa generates three problem variants, a *big-step problem*, a *small-step problem* and an *abstracted problem*, and submits each variant to both provers. A big-step problem contains the original axioms with ℓ as the conjecture, allowing the prover to find a completely new derivation. A small-step problem adds all lemmas derived before ℓ in the direct proof as additional axioms, guiding the provers toward a derivation similar to the original but potentially shorter. An abstracted problem replaces selected subterms of ℓ that do not contain nested

applications of $\diamond$ by fresh variables. For example, a subterm $x \diamond y$ might be replaced by a fresh variable z. The resulting lemma is a more general version of ℓ that may be provable with fewer steps. For each lemma, the shortest proof across all variants and both provers is retained.

The shortest proof found for each lemma determines which other lemmas it *depends on*. A lemma ℓ' is a *direct dependency* of ℓ if it appears as an axiom in the shortest proof of ℓ, and an *indirect dependency* if it is a dependency of one of ℓ's direct dependencies. Krympa builds a dependency graph for each candidate arrival lemma, capturing all direct and indirect dependencies needed to derive it.

Once the shortest proof for each lemma has been found, Krympa assembles a proof of C using a segmentation approach. The proof is divided into three segments, each proved independently by Twee or Vampire. The first runs from the axioms to a *departure lemma*, the second from the departure lemma to an *arrival lemma* and the third from the arrival lemma to C. The segmentation approach can yield shorter derivations than those found by proving lemmas individually, while keeping the search space manageable.

The three segments are formed by choosing a departure lemma and an arrival lemma. Krympa identifies up to six candidates for the arrival lemma near the end of the baseline proof, including the conjecture C itself. Different arrival lemmas typically depend on different subsets of the lemmas derived earlier in the proof, so different choices can lead to substantially different proof lengths. For each candidate arrival lemma, every lemma in its dependency graph is considered as a candidate departure lemma. For each combination of arrival and departure lemma, Krympa constructs a separate problem for each segment and submits it to both Twee and Vampire. For the first segment, the axiom A and the dependencies of the departure lemma are given as axioms, with the departure lemma as the conjecture. For the second segment, the departure lemma and its dependencies are given as axioms, with the arrival lemma as the conjecture. For the third segment, the departure lemma, the arrival lemma and their dependencies are given as axioms, with C as the conjecture. The combination yielding the shortest proof of C is retained. The final proof is output in Lean format.

Example 6. Consider a four-step baseline direct proof of $A \Rightarrow C$, where A is the axiom, C is the conjecture and L_1, L_2, L_3 are intermediate lemmas:

A	axiom
L_1	from A
L_2	from A and L_1
L_3	from L_1 and L_2
C	from L_2 and L_3

The candidate arrival lemmas are L_1, L_2, L_3 and C. As one candidate combination, consider L_3 as the arrival lemma. Its dependency graph contains L_1 and L_2, both of which are candidate departure lemmas. Taking L_1 as the departure lemma, its only dependency is the axiom A.

Krympa constructs one problem for each segment. For the first segment, A is given as the axiom with L_1 as the conjecture. For the second segment, A and L_1 are given as axioms with L_3 as the conjecture. For the third segment, A, L_1 and L_3 are given as axioms with C as the conjecture. The provers find a one-step proof of L_1, a one-step proof of L_3 directly from L_1 without needing L_2 and a one-step proof of C from L_1 and L_3, yielding the following three-step proof:

A	axiom
L_1	from A
L_3	from L_1
C	from L_1 and L_3

Applied to the 650 $\Rightarrow$ 448 problem, Krympa reduces the 62-step Vampire proof to a 20-step proof, where 13 steps are found by Twee and 7 steps by Vampire, each corresponding to a rewrite. The proof is mechanically verified in Lean. In a broader evaluation on 1431 equational theorems from the Equational Theories Project, Krympa reduces the average length of the Vampire baseline proof from 6.6 to 4.5 steps, a 31.5% reduction. The gains are largest on problems with longer baseline proofs. For the 117 problems with baseline proofs of at least 15 steps, the average decreases from 26.3 to 11.4 steps, a 56.7% reduction. Individual reductions can be dramatic. As an extreme example, a 151-step Vampire proof is reduced to 10 steps. As another extreme example, a 141-step proof is reduced to 23 steps.

4.3 Related Work

Two other researchers independently responded to Tao's challenge. Kinyon [33] produced a 24-step proof using Prover9 [41, 42]. Le Floch [39] constructed a 14-step proof by hand and formalized it in Lean. The Lean version involves 14 rewrite steps alongside additional proof terms, placing its overall length in the same range as Krympa's output.

Proof compression has been studied most extensively in the propositional setting. Stachniak [62] studied minimization in strongly finite logics. Amjad [3] and Cotton [18] developed techniques for shortening propositional resolution proofs. Vyskočil et al. [67] proposed a different angle, compressing proofs by introducing new defined symbols for recurring subterms. Krympa does not introduce new symbols but finds shorter derivations by submitting segmented subproblems to multiple provers. On the topic of combining proofs, Sutcliffe et al. [65] introduced a method for merging automatically generated proofs by representing them as directed acyclic graphs and swapping subproofs. The objective there is proof diversity rather than length reduction.

More recently, machine-learning approaches have targeted proof simplification in the Lean proof assistant. ImProver [1] uses a language model agent to rewrite Lean proofs for conciseness, and ProofOptimizer [27] trains language models to simplify Lean proofs without human demonstrations. These approaches differ from Krympa both in setting, targeting interactive proof assistants rather than output from automated theorem provers, and in method, relying on neural models rather than symbolic proof search.

5 Presenting Proofs (B.4)

This chapter addresses the third challenge described in Chapter 1: presenting proofs found by automated theorem provers in a human-readable format. Section 5.1 presents the Horn fragment of first-order logic and the readability difficulties of refutational proofs. Section 5.2 introduces Tälja, a tool we developed to translate refutational proofs into a structured, direct proof format.

5.1 Proofs for the Horn Fragment of First-Order Logic

The Horn fragment of first-order logic with equality consists of *Horn clauses*, that is, clauses with at most one positive literal, written in implicational form. A Horn clause with a positive head takes the form $B_1 \wedge \cdots \wedge B_n \to H$, where H is a single atom, either an equation $s \approx t$ or a predicate application $\mathsf{P}(t_1, \ldots, t_k)$, and $B_1, \ldots, B_n$ are atoms forming the body. A Horn clause with no positive literal takes the form $B_1 \wedge \cdots \wedge B_n \to \bot$. *Equational logic* is a subfragment of the Horn fragment, in which equality is the only predicate symbol and all clauses are unconditional equations ($n = 0$).

As discussed in Chapter 2, to prove that a conjecture follows from a set of axioms, provers typically work by refutation. They negate the conjecture, add it to the axiom set and derive $\bot$ from the extended set. This approach is efficient for proof search but creates three readability difficulties. First, the reader must mentally trace which steps depend on the negated conjecture and reconstruct how the contradiction establishes the original claim. Second, provers often omit the position at which a rewrite is applied, requiring the reader to determine this independently. Third, the low-level inference steps of the superposition calculus, combining unification and rewriting simultaneously, obscure the high-level mathematical content of the argument.

Example 7. Consider the conjecture $\mathsf{g}(\mathsf{f}(\mathsf{f}(\mathsf{b}))) \approx \mathsf{a}$ and the axioms $\mathsf{f}(x) \approx x$ and $\mathsf{g}(\mathsf{b}) \approx \mathsf{a}$. A prover based on the superposition calculus adds the negation of the conjecture to the axioms and derives a contradiction. The proof is as follows:

1.	$\mathsf{f}(x) \approx x$	axiom
2.	$\mathsf{g}(\mathsf{b}) \approx \mathsf{a}$	axiom
3.	$\mathsf{g}(\mathsf{f}(\mathsf{f}(\mathsf{b}))) \not\approx \mathsf{a}$	negated conjecture
4.	$\mathsf{g}(\mathsf{f}(\mathsf{b})) \not\approx \mathsf{a}$	by superposition from 1 and 3
5.	$\mathsf{g}(\mathsf{b}) \not\approx \mathsf{a}$	by superposition from 1 and 4
6.	$\mathsf{a} \not\approx \mathsf{a}$	by superposition from 2 and 5
7.	$\bot$	by equality resolution from 6

Step 3 is not part of the original problem. To understand the proof, the reader must keep track of which steps depend on it and reconstruct how the contradiction establishes the original claim. Steps 4 and 5 each cite a superposition inference but omit the position at which f was replaced. When verifying step 4, axiom 1 can match step 3 at two positions, the outer $\mathsf{f}(\mathsf{f}(\mathsf{b}))$ and the inner $\mathsf{f}(\mathsf{b})$, so the reader must determine which was used. Verifying steps 4 and 5 also requires unifying the left-hand side of axiom 1 with a subterm of the preceding step, mapping x to $\mathsf{f}(\mathsf{b})$ and b respectively.

5.2 Tälja: A Horn Proof Conversion Tool

We address the readability drawbacks of superposition-based refutational proofs by proposing a structured proof format for the Horn fragment. The format expresses *direct proofs*, which establish the conjecture positively. A proof consists of three parts: a list of *axioms* restating the premises, zero or more *lemmas* representing intermediate facts and a *goal* stating the conjecture to prove. Each lemma and the goal are followed by a subproof.

Each subproof takes one of two forms. An *equality chain* records a sequence of rewriting steps $t_1 \approx t_2 \approx \cdots \approx t_n$, where each step names the axiom or lemma applied and whether the rule was used left-to-right or right-to-left. A "have/hence" subproof encodes a hyperresolution step, where premises are collected one by one, the first introduced by "have" and any additional ones by "and", and the conclusion drawn by "hence" citing the axiom or lemma that combines them. A "hence" conclusion can itself open a further "have/hence" block. The format uses *matching* rather than full unification, so only the variables in the rewrite rule are instantiated, leaving the variables in the target term fixed.

Example 8. The following proof in our format establishes $\mathsf{P}(\mathsf{a})$ from five axioms via two lemmas.

Axiom 1: $\mathsf{g}(x) \approx \mathsf{f}(x)$

Axiom 2: $\mathsf{f}(x) \approx x$

Axiom 3: $\mathsf{g}(x) \approx x \to \mathsf{Q}(x)$

Axiom 4: $\mathsf{S}(\mathsf{a})$

Axiom 5: $\mathsf{S}(x) \wedge \mathsf{Q}(x) \to \mathsf{P}(x)$

Lemma 6: $\mathsf{g}(x) \approx x$
Proof:
  $\mathsf{g}(x)$
$\approx$ { by axiom 1 }
  $\mathsf{f}(x)$
$\approx$ { by axiom 2 }
  x

Lemma 7: $\mathsf{Q}(\mathsf{a})$
Proof:
  have $\mathsf{g}(\mathsf{a}) \approx \mathsf{a}$
    by lemma 6
  hence $\mathsf{Q}(\mathsf{a})$
    by axiom 3

Goal: $\mathsf{P}(\mathsf{a})$
Proof:
  have $\mathsf{S}(\mathsf{a})$
    by axiom 4
  and $\mathsf{Q}(\mathsf{a})$
    by lemma 7
  hence $\mathsf{P}(\mathsf{a})$
    by axiom 5

Lemma 6 is proved by an equality chain, in which each step rewrites a subterm and cites the axiom applied. Lemma 7 and the goal are each proved by a "have/hence" block. In the goal proof, instances of the two body atoms of axiom 5 are established separately before the concluding "hence" step.

Along with the proof format, we introduce a translation algorithm and implement it in *Tälja*, a tool that takes as input refutational proofs generated by the superposition provers E [60] and Vampire [38] as well as by the equational prover Twee [61], and converts them into direct proofs in our format.

The provers output proofs as directed acyclic graphs (DAGs). Tälja unfolds a given DAG into a proof tree, which gives a unique dependency order among clauses and determines the grounding substitution θ under which the proof holds. Tälja then classifies each clause as either an *electron*, i.e., a clause consisting of a single positive literal, or a *nucleus*, i.e., any other clause. Among nuclei, the *goal nucleus* is derived from the negated conjecture and has head $\bot$. Tälja maintains a set *Units* of established electrons, initialized with the input electrons and extended as derived electrons are processed. We illustrate how Tälja works on the following example.

Example 9. Consider the following refutational proof of $\mathsf{P}(\mathsf{a})$ from axioms 1–5, in the clausal form produced by a superposition prover:

1.	$\mathsf{g}(x) \approx \mathsf{f}(x)$	axiom
2.	$\mathsf{f}(x) \approx x$	axiom
3.	$\mathsf{g}(x) \not\approx x \vee \mathsf{Q}(x)$	axiom
4.	$\mathsf{S}(\mathsf{a})$	axiom
5.	$\mathsf{P}(x) \vee \neg\mathsf{S}(x) \vee \neg\mathsf{Q}(x)$	axiom
6.	$\neg\mathsf{P}(\mathsf{a})$	negated goal
7.	$\mathsf{g}(x) \approx x$	by superposition from 1 and 2
8.	$\neg\mathsf{S}(\mathsf{a}) \vee \neg\mathsf{Q}(\mathsf{a})$	by resolution from 5 and 6
9.	$\neg\mathsf{Q}(\mathsf{a})$	by resolution from 8 and 4
10.	$\mathsf{a} \not\approx \mathsf{g}(\mathsf{a})$	by resolution from 9 and 3
11.	$\bot$	by resolution from 10 and 7

The above proof is essentially a DAG. The corresponding proof tree is

$\bot$

- $\mathsf{g}(x) \approx x$
  - $\mathsf{g}(x) \approx \mathsf{f}(x)$ (axiom 1)
  - $\mathsf{f}(x) \approx x$ (axiom 2)
- $\mathsf{a} \not\approx \mathsf{g}(\mathsf{a})$
  - $\mathsf{g}(x) \not\approx x \vee \mathsf{Q}(x)$ (axiom 3)
  - $\neg\mathsf{Q}(\mathsf{a})$
    - $\mathsf{S}(\mathsf{a})$ (axiom 4)
    - $\neg\mathsf{S}(\mathsf{a}) \vee \neg\mathsf{Q}(\mathsf{a})$
      - $\mathsf{P}(x) \vee \neg\mathsf{S}(x) \vee \neg\mathsf{Q}(x)$ (axiom 5)
      - $\neg\mathsf{P}(\mathsf{a})$ (negated goal)

Axioms 1, 2 and 4 are input electrons. Axioms 3 and 5 and the negated goal are nuclei, with goal atom $\mathsf{P}(\mathsf{a})$. *Units* is initialized with

$$\big\{(\text{“axiom 1”}, \mathsf{g}(x) \approx \mathsf{f}(x)),\ (\text{“axiom 2”}, \mathsf{f}(x) \approx x),\ (\text{“axiom 4”}, \mathsf{S}(\mathsf{a}))\big\}$$

Tälja considers the nuclei in a specific order, processing them from left to right according to their position in the proof tree. It first processes axiom 3, then axiom 5 and finally the negated goal.

The tool starts by processing axiom 3, the nucleus $\mathsf{g}(x) \not\approx x \vee \mathsf{Q}(x)$, with body atom $\mathsf{g}(x) \approx x$ and head $\mathsf{Q}(x)$. Applying the grounding substitution θ to the body atom gives $\mathsf{g}(\mathsf{a}) \approx \mathsf{a}$. Tälja searches *Units* for an electron matching $\mathsf{g}(\mathsf{a}) \approx \mathsf{a}$, but neither axiom 1 nor axiom 2 provides a direct match. The tool then retries each candidate electron's *rewriting chain*, i.e., the sequence of rewriting steps recorded in the input proof. The input proof derives $\mathsf{g}(x) \approx x$ by rewriting axiom 1 with axiom 2, which replaces $\mathsf{f}(x)$ by x, yielding the chain $\mathsf{g}(x) \approx \mathsf{f}(x) \approx x$. Its conclusion matches the body atom of the nucleus under the substitution $x \mapsto \mathsf{a}$. Based on a readability heuristic, Tälja promotes the chain $\mathsf{g}(x) \approx \mathsf{f}(x) \approx x$ to a named lemma, lemma 6, which states $\mathsf{g}(x) \approx x$ and has the chain as its proof. The tool then assembles a

subproof for the head instance $\mathsf{Q}(\mathsf{a})$, opening with "have $\mathsf{g}(\mathsf{a}) \approx \mathsf{a}$ by lemma 6" and closing with "hence $\mathsf{Q}(\mathsf{a})$ by axiom 3". The derived electron $\mathsf{Q}(\mathsf{a})$ is added to *Units*.

Tälja then processes axiom 5, the nucleus $\mathsf{P}(x) \vee \neg\mathsf{S}(x) \vee \neg\mathsf{Q}(x)$, i.e., $\mathsf{S}(x) \wedge \mathsf{Q}(x) \rightarrow \mathsf{P}(x)$. Under the grounding substitution, its two body atoms become $\mathsf{S}(\mathsf{a})$ and $\mathsf{Q}(\mathsf{a})$. $\mathsf{S}(\mathsf{a})$ matches axiom 4 directly. $\mathsf{Q}(\mathsf{a})$ matches the derived electron $\mathsf{Q}(\mathsf{a})$. Since $\mathsf{Q}(\mathsf{a})$ is not the first body atom of the nucleus, it is promoted to a named lemma, lemma 7. Its proof is the subproof stored for $\mathsf{Q}(\mathsf{a})$, "have $\mathsf{g}(\mathsf{a}) \approx \mathsf{a}$ by lemma 6, hence $\mathsf{Q}(\mathsf{a})$ by axiom 3". The resulting proof of $\mathsf{P}(\mathsf{a})$ opens with "have $\mathsf{S}(\mathsf{a})$ by axiom 4", continues with "and $\mathsf{Q}(\mathsf{a})$ by lemma 7" and closes with "hence $\mathsf{P}(\mathsf{a})$ by axiom 5". The derived electron $\mathsf{P}(\mathsf{a})$ is added to *Units*.

Finally, Tälja processes the negated goal $\neg\mathsf{P}(\mathsf{a})$. Its body atom $\mathsf{P}(\mathsf{a})$ matches the derived electron $\mathsf{P}(\mathsf{a})$. The subproof stored for $\mathsf{P}(\mathsf{a})$, "have $\mathsf{S}(\mathsf{a})$ by axiom 4, and $\mathsf{Q}(\mathsf{a})$ by lemma 7, hence $\mathsf{P}(\mathsf{a})$ by axiom 5", is then the goal's proof. We end up with the proof presented in Example 8.

Unlike E and Vampire, Twee did not output refutational proofs in a form Tälja can process. We therefore extended Twee to produce superposition-style refutational proofs, allowing Tälja to apply the same translation to the output of all three provers.

We evaluated Tälja on 2828 benchmarks from the TPTP library [64], covering three categories that fall within the Horn fragment: *HNE* (Horn Non-Equational), containing Horn clauses without equality; *HEQ* (Horn Equational), containing Horn clauses with equality; and *UEQ* (Unit Equality), containing unit equational problems. These categories contain 821, 452 and 1140 problems, respectively. The remaining 408 FOF and 7 TFF benchmarks are theorems whose axioms and conjecture are Horn clauses. We ran each benchmark through the provers E, Twee and Vampire with a timeout of 60 seconds per benchmark. E proved 2270 problems, all of which Tälja successfully translated. Twee proved 1739 problems, of which Tälja translated 1737; for the remaining 2, Twee found a proof but failed to output it as a refutation. Vampire proved 2068 problems, of which Tälja translated 2067. To verify correctness, Tälja can also output the proof as a Lean [47] file. All translated proofs were accepted by the Lean kernel.

5.3 Related Work

Our proof format and translation algorithm build directly on the work of Denzinger and Schulz [20], who introduced a readable output format for the DISCOUNT prover. Their format represents proofs as sequences of lemmas with equational chains as justifications. Both Twee and Waldmeister [29] adopted this format. Denzinger and Schulz additionally developed heuristics to decide which intermediate steps to name as lemmas. In Twee, the rule is simple: any step cited more than once becomes a lemma. We extend this work to the full Horn fragment by adding conditional clauses and the "have/hence" connective.

Much earlier work on readable proof output for first-order logic without equality uses natural deduction as the target format. Natural deduction is widely regarded as close to how mathematicians reason, although it tends to produce long derivations.

Miller and Felty [44] gave an early account combining resolution refutations with natural deduction derivations. Lingenfelder [40] gave a translation from refutation graphs to natural deduction. In particular, Huang [30, 31] addressed the verbosity problem by introducing assertions, which collapse routine proof steps into single derivation steps. He showed how to produce assertion-level proofs from both natural deduction proofs and resolution proofs. Meier's TRAMP system [43] later implemented this resolution-to-assertion-level translation. In the propositional setting, Amjad [2] gave a size-reducing translation from resolution proofs to natural deduction. Moreover, De Nivelle [52] tackled the problem of Skolem symbols by translating resolution proofs into natural deduction while removing them, which helps with proof checking in proof assistants that lack a choice axiom. Similar work has been done in higher-order logic. Andrews [5] showed that refutational proofs can be cast in natural deduction style, and the TPS prover [6] used this to present its results in a readable form.

Another approach is taken by the GAPT system [23]. Rather than translating the resolution proof structure directly, GAPT converts the proof into an expansion proof that records the ground instantiations used, and then uses those instantiations to reconstruct a sequent calculus proof. Additionally, Ebner and Schlaipfer [24] translated such sequent calculus proofs further into natural deduction. Ebner et al. [22] instead decomposed the ground instantiations to introduce quantified lemmas. Moreover, Ebner [21] showed that a tableau-style reconstruction from the expansion proof is guaranteed to succeed for Horn clause problems, producing intuitionistic proofs in that case.

Many, if not most, automated theorem provers produce proofs in the TSTP format [64]. This format uses a standard machine-readable syntax for exchanging proofs between tools, but is not optimized for human readers.

Proof translations have also been studied from a proof complexity angle, independently of their use in theorem provers. Cook and Reckhow [17] introduced polynomial simulation between propositional proof systems as a way to compare their efficiency. In particular, Nordström [54] surveyed how such translations and the resulting lower bounds directly constrain the runtime of resolution-based SAT solvers.

6 Conclusion and Future Directions

This thesis addressed the challenges of finding, simplifying and presenting proofs automatically in first- and higher-order logic. For finding proofs, we introduced MBQI-Enum, a quantifier instantiation strategy, integrated into the SMT solver cvc5 [11], which augments model-based quantifier instantiation with a syntax-guided grammar enumerator. For simplifying proofs, we introduced Krympa, a tool that decomposes equational proofs into lemmas, re-proves each lemma independently using Twee and Vampire, and reassembles the results using heuristics to obtain shorter derivations. For presenting proofs, we introduced Tälja, a tool that translates refutational resolution and superposition proofs for the Horn fragment of first-order logic into direct "have/hence" proofs. Overall, this thesis showed that proof automation and readability can go hand in hand.

There are several directions for future work. Concerning finding proofs, future work includes investigating further quantifier instantiation strategies for higher-order logic. One promising candidate is counterexample-guided instantiation, which derives new instances from counterexamples to the current candidate model rather than enumerating them from a grammar, and could complement the approach taken by MBQI-Enum. Concerning simplifying proofs, a natural extension of Krympa would be to move beyond equational logic to the full Horn fragment and, eventually, to higher-order logic, adapting the decomposition and segmentation strategies to handle conditional clauses and λ-terms. Similarly, Tälja is currently restricted to the Horn fragment. Extending it to full first-order logic would require a more expressive proof format, whereas a further extension to higher-order logic would enable it to produce readable proofs for the problems MBQI-Enum targets. Another direction would be to integrate Krympa into Tälja, using Krympa to shorten the equational proofs that appear as subproofs within Tälja's output.

Bibliography

[1] Riyaz Ahuja, Jeremy Avigad, Prasad Tetali, and Sean Welleck. "ImProver: Agent-Based Automated Proof Optimization". In: *ICLR 2025*. OpenReview.net, 2025. URL: https://openreview.net/forum?id=dWsdJAXjQD.

[2] Hasan Amjad. "A Compressing Translation from Propositional Resolution to Natural Deduction". In: *FroCoS 2007*. Ed. by Boris Konev and Frank Wolter. Vol. 4720. LNCS. Springer, 2007, pp. 88–102. DOI: 10.1007/978-3-540-74621-8_6.

[3] Hasan Amjad. "Compressing Propositional Refutations". In: *AVoCS 2006*. Ed. by Stephan Merz and Tobias Nipkow. Vol. 185. Electronic Notes in Theoretical Computer Science. Elsevier, 2006, pp. 3–15. DOI: 10.1016/J.ENTCS.2007.05.025.

[4] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, 1986. ISBN: 978-0-12-058535-9.

[5] Peter B. Andrews. "Transforming Matings into Natural Deduction Proofs". In: *CADE 1980*. Ed. by Wolfgang Bibel and Robert A. Kowalski. Vol. 87. LNCS. Springer, 1980, pp. 281–292. DOI: 10.1007/3-540-10009-1_22.

[6] Peter B. Andrews and Chad E. Brown. "TPS: A hybrid automatic-interactive system for developing proofs". In: *Journal of Applied Logic* 4.4 (2006), pp. 367–395. DOI: 10.1016/J.JAL.2005.10.002.

[7] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. "Completion Without Failure". In: *Rewriting Techniques*. Ed. by Hassan Aït-Kaci and Maurice Nivat. Academic Press, 1989, pp. 1–30. DOI: 10.1016/B978-0-12-046371-8.50007-9.

[8] Leo Bachmair and Harald Ganzinger. "Resolution Theorem Proving". In: *Handbook of Automated Reasoning*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 19–99. DOI: 10.1016/B978-044450813-3/50004-7.

[9] Leo Bachmair and Harald Ganzinger. "Rewrite-Based Equational Theorem Proving with Selection and Simplification". In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247. DOI: 10.1093/LOGCOM/4.3.217.

[10] Julian Backes and Chad E. Brown. "Analytic Tableaux for Higher-Order Logic with Choice". In: *Journal of Automated Reasoning* 47.4 (2011), pp. 451–479. DOI: 10.1007/S10817-011-9233-2.

[11] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *TACAS 2022*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. LNCS. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[12] Clark W. Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[13] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. "Superposition for Higher-Order Logic". In: *Journal of Automated Reasoning* 67.1 (2023), p. 10. DOI: 10.1007/S10817-022-09649-9.

[14] Ahmed Bhayat and Martin Suda. "A Higher-Order Vampire (Short Paper)". In: *IJCAR 2024*. Ed. by Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt. Vol. 14739. LNCS. Springer, 2024, pp. 75–85. DOI: 10.1007/978-3-031-63498-7_5.

[15] Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, Floris van Doorn, Martin Dvorak, Andrés Goens, Aaron Hill, Harald Husum, Hernán Ibarra Mejia, Zoltan A. Kocsis, Bruno Le Floch, Amir Livne Bar-on, Lorenzo Luccioli, Douglas McNeil, Alex Meiburg, Pietro Monticone, Pace P. Nielsen, Emmanuel Osalotioman Osazuwa, Giovanni Paolini, Marco Petracci, Bernhard Reinke, David Renshaw, Marcus Rossel, Cody Roux, Jérémy Scanvic, Shreyas Srinivas, Anand Rao Tadipatri, Terence Tao, Vlad Tsyrklevich, Fernando Vaquerizo-Villar, Daniel Weber, and Fan Zheng. "The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale". In: *CoRR* abs/2512.07087 (2025). DOI: 10.48550/ARXIV.2512.07087.

[16] Chad E. Brown. "Satallax: An Automatic Higher-Order Prover". In: *IJCAR 2012*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. LNCS. Springer, 2012, pp. 111–117. DOI: 10.1007/978-3-642-31365-3_11.

[17] Stephen A. Cook and Robert A. Reckhow. "The Relative Efficiency of Propositional Proof Systems". In: *Journal of Symbolic Logic* 44.1 (1979), pp. 36–50. DOI: 10.2307/2273702.

[18] Scott Cotton. "Two Techniques for Minimizing Resolution Proofs". In: *SAT 2010*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. LNCS. Springer, 2010, pp. 306–312. DOI: 10.1007/978-3-642-14186-7_26.

[19] Marek Danco, Petra Hozzová, and Mikolás Janota. "From MBQI to Enumerative Instantiation and Back". In: *SMT 2025*. Ed. by Jochen Hoenicke, Mikolás Janota, Aina Niemetz, and Sophie Tourret. Vol. 4008. CEUR Workshop Proceedings. CEUR-WS.org, 2025, pp. 52–58. URL: https://ceur-ws.org/Vol-4008/SMT_paper10.pdf.

[20] Jörg Denzinger and Stephan Schulz. "Recording, Analyzing and Presenting Distributed Deduction Processes". In: *PASCO 1994*. Ed. by Hoon Hong. World Scientific, 1994, pp. 114–123. DOI: 10.1142/9789814533584_0012.

[21] Gabriel Ebner. "Herbrand Constructivization for Automated Intuitionistic Theorem Proving". In: *TABLEAUX 2019*. Ed. by Serenella Cerrito and Andrei Popescu. Vol. 11714. LNCS. Springer, 2019, pp. 355–373. DOI: 10.1007/978-3-030-29026-9_20.

[22] Gabriel Ebner, Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. "On the Generation of Quantified Lemmas". In: *Journal of Automated Reasoning* 63.1 (2019), pp. 95–126. DOI: 10.1007/S10817-018-9462-8.

[23] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. "System Description: GAPT 2.0". In: *IJCAR 2016*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. LNCS. Springer, 2016, pp. 293–301. DOI: 10.1007/978-3-319-40229-1_20.

[24] Gabriel Ebner and Matthias Schlaipfer. "Efficient Translation of Sequent Calculus Proofs Into Natural Deduction Proofs". In: *PAAR 2018*. Ed. by Boris Konev, Josef Urban, and Philipp Rümmer. Vol. 2162. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 17–33. URL: https://ceur-ws.org/Vol-2162/paper-03.pdf.

[25] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "DPLL(T): Fast Decision Procedures". In: *CAV 2004*. Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. LNCS. Springer, 2004, pp. 175–188. DOI: 10.1007/978-3-540-27813-9_14.

[26] Yeting Ge and Leonardo Mendonça de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *CAV 2009*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. LNCS. Springer, 2009, pp. 306–320. DOI: 10.1007/978-3-642-02658-4_25.

[27] Alex Gu, Bartosz Piotrowski, Fabian Gloeckle, Kaiyu Yang, and Aram H. Markosyan. "ProofOptimizer: Training Language Models to Simplify Proofs without Human Demonstrations". In: *CoRR* abs/2510.15700 (2025). DOI: 10.48550/ARXIV.2510.15700.

[28] Leon Henkin. "Completeness in the Theory of Types". In: *Journal of Symbolic Logic* 15.2 (1950), pp. 81–91. DOI: 10.2307/2266967.

[29] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. "WALDMEISTER - High-Performance Equational Deduction". In: *Journal of Automated Reasoning* 18.2 (1997), pp. 265–270. DOI: 10.1023/A:1005872405899.

[30] Xiaorong Huang. "Reconstruction Proofs at the Assertion Level". In: *CADE 1994*. Ed. by Alan Bundy. Vol. 814. LNCS. Springer, 1994, pp. 738–752. DOI: 10.1007/3-540-58156-1_53.

[31] Xiaorong Huang. "Translating Machine-Generated Resolution Proofs into ND-Proofs at the Assertion Level". In: *PRICAI 1996*. Ed. by Norman Y. Foo and Randy Goebel. Vol. 1114. LNCS. Springer, 1996, pp. 399–410. DOI: 10.1007/3-540-61532-6_34.

[32] Jan Jakubuv and Mikolás Janota. "Quantifier Instantiations: To Mimic or To Revolt". In: *SMT 2025*. Ed. by Jochen Hoenicke, Mikolás Janota, Aina Niemetz, and Sophie Tourret. Vol. 4008. CEUR Workshop Proceedings. CEUR-WS.org, 2025, pp. 45–51. URL: https://ceur-ws.org/Vol-4008/SMT_paper08.pdf.

[33] Michael Kinyon. *Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)"*. 2025. URL: https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518961204.

[34] Lydia Kondylidou, Jasmin Blanchette, and Marijn J. H. Heule. "Tao's Equational Proof Challenge Accepted". In: *IJCAR 2026*. Ed. by Armin Biere, Carsten Lutz, and Sara Negri. Vol. 16688. LNCS. Springer, 2026, pp. 3–21. DOI: 10.1007/978-3-032-32589-1_1.

[35] Lydia Kondylidou, Jasmin Blanchette, Nicholas Smallbone, and Uwe Waldmann. "Readable Proofs for the Horn Fragment". 2026.

[36] Lydia Kondylidou, Andrew Reynolds, and Jasmin Blanchette. "Augmenting Model-Based Instantiation with Fast Enumeration". In: *TACAS 2025*. Ed. by Arie Gurfinkel and Marijn Heule. Vol. 15696. LNCS. Springer, 2025, pp. 85–103. DOI: 10.1007/978-3-031-90643-5_5.

[37] Lydia Kondylidou, Andrew Reynolds, Jasmin Blanchette, and Cesare Tinelli. "Enumerating Choice Terms in Model-Based Quantifier Instantiation". In: *TACAS 2026*. Ed. by Sebastian Junges and Guy Katz. Vol. 16505. LNCS. Springer, 2026, pp. 295–314. DOI: 10.1007/978-3-032-22752-2_15.

[38] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *CAV 2013*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Springer, 2013, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

[39] Bruno Le Floch. *Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)"*. 2025. URL: https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518970125.

[40] Christoph Lingenfelder. "Structuring Computer Generated Proofs". In: *IJCAI 1989*. Ed. by N. S. Sridharan. Morgan Kaufmann, 1989, pp. 378–383. URL: http://ijcai.org/Proceedings/89-1/Papers/060.pdf.

[41] Jeffrey P. Machado and Larry Lesyna. *Enhanced LADR-2026*. 2026. URL: https://prover9.org.

[42] William McCune. *Prover9 and Mace4*. 2009. URL: https://www.cs.unm.edu/~mccune/prover9/.

[43] Andreas Meier. "System Description: TRAMP: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level". In: *CADE 2000*. Ed. by David A. McAllester. Vol. 1831. LNCS. Springer, 2000, pp. 460–464. DOI: 10.1007/10721959_37.

[44] Dale Miller and Amy P. Felty. "An Integration of Resolution and Natural Deduction Theorem Proving". In: *AAAI 1986*. Ed. by Tom Kehler. Morgan Kaufmann, 1986, pp. 198–202. URL: http://www.aaai.org/Library/AAAI/1986/aaai86-032.php.

[45] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Efficient E-Matching for SMT Solvers". In: *CADE 2007*. Ed. by Frank Pfenning. Vol. 4603. LNCS. Springer, 2007, pp. 183–198. DOI: 10.1007/978-3-540-73595-3_13.

[46] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS 2008*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[47] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *CADE 2015*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. LNCS. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

[48] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Transactions on Programming Languages and Systems* 1.2 (1979), pp. 245–257. DOI: 10.1145/357073.357079.

[49] Aina Niemetz and Mathias Preiner. "Bitwuzla". In: *CAV 2023*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. LNCS. Springer, 2023, pp. 3–17. DOI: 10.1007/978-3-031-37703-7_1.

[50] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. "Syntax-Guided Quantifier Instantiation". In: *TACAS 2021*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. LNCS. Springer, 2021, pp. 145–163. DOI: 10.1007/978-3-030-72013-1_8.

[51] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. "Btor2 , BtorMC and Boolector 3.0". In: *CAV 2018*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. LNCS. Springer, 2018, pp. 587–595. DOI: 10.1007/978-3-319-96145-3_32.

[52] Hans de Nivelle. "Translation of resolution proofs into short first-order proofs without choice axioms". In: *Information and Computation* 199.1-2 (2005), pp. 24–54. DOI: 10.1016/J.IC.2004.10.011.

[53] Andreas Nonnengart and Christoph Weidenbach. "Computing Small Clause Normal Forms". In: *Handbook of Automated Reasoning*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 335–367. DOI: 10.1016/B978-044450813-3/50008-4.

[54] Jakob Nordström. "On the interplay between proof complexity and SAT solving". In: *ACM SIGLOG News* 2.3 (2015), pp. 19–44. DOI: 10.1145/2815493.2815497.

[55] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. "Revisiting Enumerative Instantiation". In: *TACAS 2018*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. LNCS. Springer, 2018, pp. 112–131. DOI: 10.1007/978-3-319-89963-3_7.

[56] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. "cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis". In: *CAV 2019*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. LNCS. Springer, 2019, pp. 74–83. DOI: 10.1007/978-3-030-25543-5_5.

[57] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *CAV 2015*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. LNCS. Springer, 2015, pp. 198–216. DOI: 10.1007/978-3-319-21668-3_12.

[58] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. "Finding conflicting instances of quantified formulas in SMT". In: *FMCAD 2014*. IEEE, 2014, pp. 195–202. DOI: 10.1109/FMCAD.2014.6987613.

[59] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *Journal of the ACM* 12.1 (1965), pp. 23–41. DOI: 10.1145/321250.321253.

[60] Stephan Schulz. "E - a brainiac theorem prover". In: *AI Communications* 15.2-3 (2002), pp. 111–126. URL: http://content.iospress.com/articles/ai-communications/aic260.

[61] Nicholas Smallbone. "Twee: An Equational Theorem Prover". In: *CADE 2021*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. LNCS. Springer, 2021, pp. 602–613. DOI: 10.1007/978-3-030-79876-5_35.

[62] Zbigniew Stachniak. "Minimization of resolution proof systems". In: *Fundamenta Informaticae* 14.1 (1991), pp. 129–146. DOI: 10.3233/fi-1991-14107.

[63] Alexander Steen. "Extensional paramodulation for higher-order logic and its effective implementation Leo-III". PhD thesis. Free University of Berlin, Dahlem, Germany, 2018. URL: https://d-nb.info/1167441656.

[64] Geoff Sutcliffe. "Stepping Stones in the TPTP World". In: *IJCAR 2024*. Ed. by Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt. Vol. 14739. LNCS. Springer, 2024, pp. 30–50. DOI: 10.1007/978-3-031-63498-7_3.

[65] Geoff Sutcliffe, Cynthia Chang, Deborah L. McGuinness, Timothy Lebo, Li Ding, and Paulo Pinheiro da Silva. "Combining Proofs to form Different Proofs". In: *PxTP 2011*. Ed. by Pascal Fontaine and Aaron Stump. 2011, pp. 60–73. URL: https://pxtp2011.loria.fr/PxTP2011.pdf%5C#page=64.

[66] Terence Tao. *Zulip thread: Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)*. 2025. URL: https://leanprover-community.github.io/archive/stream/219941-Machine-Learning-for-Theorem-Proving/.

[67] Jirí Vyskocil, David Stanovský, and Josef Urban. "Automated Proof Compression by Invention of New Definitions". In: *LPAR 2010*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. LNCS. Springer, 2010, pp. 447–462. DOI: 10.1007/978-3-642-17511-4_25.

A Credits

Augmenting Model-Based Instantiation with Fast Enumeration

This article (B.1) is authored by Lydia Kondylidou, Andrew Reynolds and Jasmin Blanchette and published by Springer in the proceedings of TACAS 2025 [36]. With guidance from Jasmin Blanchette, Lydia Kondylidou identified that existing quantifier instantiation strategies struggle with higher-order quantifiers, motivating this work, and developed the examples illustrating the approach. Andrew Reynolds designed the algorithmic framework for integrating the enumerator into the instantiation loop. The implementation in the SMT solver cvc5 was carried out jointly by Andrew Reynolds and Lydia Kondylidou. The core of the paper was written primarily by Lydia Kondylidou, with co-authors contributing smaller sections, along with guidance, feedback and revisions. Lydia Kondylidou is the main author of this article.

Enumerating Choice Terms in Model-Based Quantifier Instantiation

This article (B.2) is authored by Lydia Kondylidou, Andrew Reynolds, Jasmin Blanchette and Cesare Tinelli and published by Springer in the proceedings of TACAS 2026 [37]. Lydia Kondylidou identified that MBQI-Enum fails to find refutations when Hilbert choice terms are needed, and recognized this limitation as the central problem to address. The core extension was developed together with Jasmin Blanchette. The implementation in cvc5 was carried out jointly by Andrew Reynolds and Lydia Kondylidou. Cesare Tinelli provided guidance when theoretical and practical difficulties arose. The core of the paper was written primarily by Lydia Kondylidou; co-authors contributed smaller sections, along with guidance, feedback and revisions. Lydia Kondylidou is the main author of this article.

Tao's Equational Proof Challenge Accepted

This article (B.3) is authored by Lydia Kondylidou, Jasmin Blanchette and Marijn J.H. Heule and published by Springer in the proceedings of IJCAR 2026 [34]. Marijn Heule brought to Lydia Kondylidou's attention the challenge posed by Terence Tao of finding a shorter proof for a specific equational problem than the 62-step proof found by the Vampire prover. The idea of decomposing a proof into lemmas and minimizing each independently was proposed by Jasmin Blanchette. Lydia Kondylidou

designed and implemented the tool, including the overall architecture, the minimization strategies and the heuristics, and conducted the empirical evaluation. Jasmin Blanchette and Marijn Heule provided guidance throughout. The core of the paper was written primarily by Lydia Kondylidou; co-authors contributed smaller sections, along with guidance, feedback and revisions. Lydia Kondylidou is the main author of this article.

Readable Proofs for the Horn Fragment

This article (B.4) is authored by Lydia Kondylidou, Jasmin Blanchette, Nicholas Smallbone and Uwe Waldmann and is to be submitted to CPP 2027. The idea originated as future work suggested in Paper B.3 by Jasmin Blanchette and Lydia Kondylidou. Together, they created initial mockup examples to shape the design of the format. Lydia Kondylidou designed the proof format. The translation algorithm from refutational proofs into the structured format was developed jointly by Lydia Kondylidou and Uwe Waldmann. Uwe Waldmann also proved its correctness. Lydia Kondylidou implemented the translation tool. Nicholas Smallbone implemented the extension of Twee to output refutational proofs. The work was carried out under the ongoing guidance of Jasmin Blanchette. The paper was written primarily by Lydia Kondylidou. Uwe Waldmann wrote the correctness proof subsection. Nicholas Smallbone wrote the integration with Twee section. Jasmin Blanchette wrote the background section and provided feedback throughout. Lydia Kondylidou is the main author of this article.

B Papers

This appendix includes all papers discussed in Chapters 3, 4, and 5.







Augmenting Model-Based Instantiation with Fast Enumeration

Lydia Kondylidou[1(✉)], Andrew Reynolds[2], and Jasmin Blanchette[1]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
{l.kondylidou,jasmin.blanchette}@lmu.de
[2] The University of Iowa, Iowa City, USA
andrew.j.reynolds@gmail.com

Abstract. Satisfiability modulo theories (SMT) solvers rely on various quantifier instantiation strategies to support first- and higher-order logic. We introduce MBQI-Enum, an approach that extends model-based quantifier instantiation (MBQI) with syntax-guided synthesis (SyGuS) techniques. Our approach targets first-order theories without well-established quantifier instantiation techniques and higher-order quantifiers that can benefit from instantiations with λ-terms. By incorporating a SyGuS enumerator, our approach generates a broader set of candidate instantiations, including identity functions and terms containing uninterpreted symbols, thereby improving the effectiveness of MBQI.

1 Introduction

Satisfiability modulo theories (SMT) solvers combine a Boolean satisfiability (SAT) solver with decision procedures for interpreted theories. Several SMT solvers, including Bitwuzla [16], Boolector [17], cvc5 [2], veriT [7], and Z3 [15], also support quantifiers via Skolemization and instantiation. With complete instantiation strategies, SMT solvers offer a semidecision procedure for first-order logic with theories. SMT has also been partly extended to higher-order logic [4].

Quantifier instantiation is a technique whereby quantified variables in a formula are instantiated with ground terms until a contradiction is found or all necessary instantiations have been generated. Consider the unsatisfiable axiom $(\forall x.\ \mathsf{p}\ x) \wedge \neg\, \mathsf{p}\ \mathsf{b}$. The SAT solver first finds a model that makes $\forall x.\ \mathsf{p}\ x$ true and $\mathsf{p}\ \mathsf{b}$ false, taken as black boxes. Then the quantifier instantiation strategy of the SMT solver might heuristically instantiate x with a, resulting in the formula $(\forall x.\ \mathsf{p}\ x) \Longrightarrow \mathsf{p}\ \mathsf{a}$, which is conjoined with the axiom. Next, the SAT solver finds a new model that also makes $\mathsf{p}\ \mathsf{a}$ true. At this point, the instantiation strategy might instantiate x with b, resulting in $(\forall x.\ \mathsf{p}\ x) \Longrightarrow \mathsf{p}\ \mathsf{b}$. Since the SAT solver cannot make $\mathsf{p}\ \mathsf{b}$ both true and false at the same time, the axiom conjoined with the two additional formulas is unsatisfiable at the SAT level, meaning that the original axiom is unsatisfiable in the SMT logic.

One successful instantiation strategy is model-based quantifier instantiation (MBQI) [9]. Briefly, it iteratively refines a candidate model constructed from the quantifier-free part of the problem. This model guides the generation of new

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15696, pp. 85–103, 2025.
https://doi.org/10.1007/978-3-031-90643-5_5

terms for instantiating quantifiers, reducing the search space. MBQI is complete for certain fragments and tends to generate small models for satisfiable problems. But it also has some limitations: First, MBQI instantiates quantifiers only with terms that denote values in a theory. In particular, it does not consider the problem's uninterpreted symbols when creating instantiations, leading it to miss some useful instantiations. Second, for higher-order problems, MBQI cannot generate function terms that return an argument, such as the identity $\lambda x.\ x$.

Another instantiation strategy, which addresses these limitations, is syntax-guided instantiation (SyQI) [18]. This constructs a grammar for each universally quantified variable according to its type and uses enumerated terms derived from the grammar as instantiations. Its main weakness is that although it is model-based like MBQI, it uses a less scalable approach to refining models that centers around syntactic constraints.

In this paper, we propose a new strategy, MBQI-Enum, that combines the strengths of MBQI and SyQI. Specifically, our strategy augments the set of instantiations created from the MBQI model with instantiations generated by a syntax-guided synthesis (SyGuS) grammar. It incorporates uninterpreted symbols gathered from the problem. For higher-order problems, it also considers λ-abstractions as potential instantiations. In this way, we exploit the fast model-finding capabilities of MBQI and the diversity of terms considered by SyQI.

Our work shares some similarities with Preiner et al. [21], but their approach was limited to selected first-order theories and did not handle higher-order logic. Our approach is resolutely pragmatic; it does not aim at completeness, which can be achieved using other instantiation strategies.

As an example where λ-terms are needed, consider the unsatisfiable problem consisting of the axiom $\forall z.\exists x,y.\ z\ x\ y \neq x$, where z ranges over binary functions. Running cvc5 with strategies such as MBQI and higher-order E-matching [4] leads the solver to give up early. By contrast, with our strategy, cvc5 immediately finds a contradiction based on the substitution $\{z \mapsto \lambda x,\ y.\ x\}$. Indeed, if we instantiate z with $\lambda x,\ y.\ x$ in the axiom and β-reduce, we obtain $x \neq x$.

We implemented MBQI-Enum in cvc5. Our empirical evaluation finds that the strategy increases the number of solved problems for a benchmark suite consisting of various first-order SMT-LIB files by a noticeable margin. Our approach is especially useful to solve unsatisfiable problems involving theories without well-established quantifier instantiation techniques. We can also report substantial gains on higher-order TPTP benchmarks. The raw evaluation data are publicly available online.[3] Our source code, along with instructions for reproducing the experiments, is also available online.[4]

2 Preliminaries

Our work relies on the following pillars: higher-order logic, SMT with quantifiers, MBQI, and SyQI.

[3] https://doi.org/10.5281/zenodo.14627782
[4] https://doi.org/10.5281/zenodo.14604430

Higher-Order Logic. Monomorphic higher-order logic [1,10], also called simple type theory [8], generalizes classical first-order logic by allowing quantification over functions. The syntax distinguishes between types and terms. Types τ are either base types κ or applications of the function arrow $\to$ to two types: $\tau_1 \to \tau_2$. The type of Booleans is denoted by o.

The term language is based on Church's simply typed λ-calculus, where terms t, e are inductively defined as variables w, x, y, z, ..., function symbols p, f, constants a, b, c, term applications $t\ t'$, and λ-abstractions $\lambda x.\ t$, where x is the bound variable and t is the body. A variable is free in a term if it is not bound by a λ-abstraction. We let $\bar{x}$ stand for $x_1, x_2, \ldots, x_n$, where $n \geq 1$. Terms are syntactically equal modulo α-, β-, and η-conversion, meaning, for example, that $(\lambda x.\ x)\ \mathsf{c}$ is equal to c. Throughout this work, we assume that terms are expressed in η-long β-normal form, meaning that η-expansion and β-reduction have been applied exhaustively. The type of a term t is of the form $\tau_1 \to \cdots \to \tau_n \to \tau$, where τ_i are the argument types and τ is the result type. If τ is of type o, which is the distinguished Boolean type, we call t a predicate. A term of type o is called a formula.

SMT with Quantifiers. Traditionally, SMT solvers work on problems in first-order logic, but they have partly been extended to higher-order logic [4], and in this paper we consider both first- and higher-order problems. SMT solvers that support quantifiers typically do so via a combination of Skolemization and instantiation: Universal quantifiers occurring negatively can be Skolemized; universal quantifiers occurring positively are instantiated heuristically; and existential quantifiers are expressed in terms of $\forall$ using the equivalence $(\exists x.\ \phi) \iff \neg(\forall x.\ \neg\phi)$. To simplify the presentation, we will assume that formulas are in a normal form where possibly negated $\forall$-quantifiers all appear in a cluster at the top level—e.g., $\forall x, y.\ \neg\, \forall z.\ \mathsf{p}\ x\ y\ z$.

Let $\mathcal{T}$ be a theory for a set of interpreted symbols, and let F be the input formula over $\mathcal{T}$. The goal is to find a model of F or derive a contradiction. If F does not contain quantifiers, the quantifier-free part of the SMT solver searches for a $\mathcal{T}$-satisfiable set L of literals that propositionally satisfies F. If such an L exists, F is $\mathcal{T}$-satisfiable (i.e., satisfiable with respect to $\mathcal{T}$); otherwise, F is $\mathcal{T}$-unsatisfiable.

If the formula F contains quantifiers, the quantifier-free solver cannot be directly applied. In the SMT solver's main loop, presented in Algorithm 1, the SMT solver tries to find a set L of literals whose atoms come from F and that propositionally satisfies F. Briefly, L is partitioned into a set E of ground literals and two sets of quantified literals, Q_p and Q_n. The ground literals within E must be $\mathcal{T}$-satisfiable, Q_p consists of formulas of the form $\forall\bar{x}.\ \phi$, and Q_n consists of formulas of the form $\neg\,\forall\bar{x}.\phi$. If the solver is unable to find such an L, it concludes that the formula F is $\mathcal{T}$-unsatisfiable.

On the other hand, if a set L of literals is found, new lemmas A are generated through the instantiation and Skolemization rounds. The instantiation round generates *instantiation lemmas*—that is, lemmas of the form $q_i \Longrightarrow q_i\sigma$—from the sets Q_p and E. For each $q_i \in Q_\mathrm{p}$, an instantiation strategy computes

Algorithm 1 The main SMT loop

1: **function** SMT-LOOP(F)
2: find a set of literals L where
3: — $L \models_{\mathrm{p}} F$, where L's atoms are a subset of atoms(F)
4: — L can be partitioned into $(E, Q_{\mathrm{p}}, Q_{\mathrm{n}})$, where
5: — E is ground and $\mathcal{T}$-satisfiable
6: — Q_{p} consists of universally quantified atoms
7: — Q_{n} consists of negated universally quantified atoms
8: **if** no such L exists **then**
9: **return** unsat
10: $A \leftarrow$ INSTANTIATION-ROUND$(Q_{\mathrm{p}}, E) \cup$ SKOLEMIZATION-ROUND(Q_{n})
11: **if** $A = \emptyset$ **then**
12: **return** sat
13: **else**
14: **return** SMT-LOOP$(F \cup A)$
15: **function** INSTANTIATION-ROUND$(\{q_1, \ldots, q_{\mathrm{n}}\}, E)$
16: $I \leftarrow \emptyset$
17: **for each** $i \in \{1, \ldots, n\}$ **do**
18: $I \leftarrow I \cup \{q_i \Longrightarrow q_i\sigma \mid \sigma \in \text{INSTS}(q_i, E)\}$
19: **return** I
20: **function** INSTS(q_i, E)
21: **return** a set of substitutions for q_i based on (q_i, E)
22: **function** SKOLEMIZATION-ROUND$(\{\neg q_1, \ldots, \neg q_{\mathrm{n}}\})$
23: $K \leftarrow \emptyset$
24: **for each** $i \in \{1, \ldots, n\}$ **do**
25: $K \leftarrow K \cup \{\neg q_i \Longrightarrow \neg q_i\sigma_{k,i}\}$, where $\sigma_{k,i}$ maps to Skolem constants for q_i
26: **return** K

substitutions σ for every top-level variable in q_i mapping them to terms. The Skolemization round generates *Skolemization lemmas* from the set Q_{n}—that is, lemmas of the form $\neg q_i \Longrightarrow \neg q_i\sigma_{j,i}$, where $\sigma_{j,i}$ instantiates every top-level variable in q_i with a Skolem constant. (We abuse notation and write $(\forall \bar{x}.\ \phi)\sigma$ to mean $\phi\sigma$, syntactically conflating quantifier instantiation and substitution.) The lemmas are then added to the original formula F, and the SMT loop is called recursively. On line 12 of Algorithm 1, we assume that the instantiation strategy denoted by INSTS is *model-sound*, meaning that if INSTS returns an empty set of substitutions, it indicates that the formula F is $\mathcal{T}$-satisfiable.

We now define the instantiation strategy INSTS starting with a naive approach. Subsequently, we will introduce more advanced techniques, including MBQI and SyQI, followed by our strategy, MBQI-Enum. These strategies can replace the naive approach to improve the solver's efficiency.

Definition 2. An *instantiation strategy* takes as input a set of ground terms E and a quantified formula q of the form $\forall \bar{x}.\ \phi$, and outputs a set of grounding substitutions $\{\sigma_1, \ldots, \sigma_m\}$, where the variables mapped by σ_i are exactly the variables in $\bar{x}$ for each $i \in \{1, \ldots, m\}$.

Example 3. Let $\mathsf{a} : \mathit{Int}$ and $\mathsf{p} : \mathit{Int} \to \mathit{Bool}$. Let F be the formula $(\forall y.\ \mathsf{p}\ y) \wedge \neg\, \mathsf{p}\ \mathsf{a}$, where $y : \mathit{Int}$. In the SMT loop, the sets $E = \{\neg\, \mathsf{p}\ \mathsf{a}\}$, $Q_\mathrm{p} = \{\forall y.\ \mathsf{p}\ y\}$, and $Q_\mathrm{n} = \emptyset$ are defined. The naive instantiation strategy produces substitutions mapping y to terms from E of the same type as y, as shown below:

$q \in Q_\mathrm{p}$	$\textsc{insts}(E, q)$
$\forall y.\ \mathsf{p}\ y$	$\{\{y \mapsto \mathsf{a}\}\}$

The instantiation lemma $(\forall y.\mathsf{p}\ y) \Longrightarrow \mathsf{p}\ \mathsf{a}$ is added to F. Now, the quantifier-free solver finds a contradiction.

The most widely used strategy for quantifier instantiation is *E-matching* [14]. This is a heuristic and typically incomplete technique that chooses substitutions by matching ground terms with *patterns*. In Example 3, the ground term $\mathsf{p}\ \mathsf{a}$ matches $\mathsf{p}\ y$ for substitution $\{y \mapsto \mathsf{a}\}$, and hence this substitution is returned by the strategy. Over the past decade, SMT solvers have been extended with more sophisticated approaches. *Conflict-based instantiation* [26] is another incomplete technique that attempts to find a single instantiation that would induce a ground conflict, before resorting to E-matching. *MBQI* [9] is a technique for finding instantiations that refute a candidate model, and is typically run when E-matching saturates. This strategy is also able to answer "$\mathcal{T}$-satisfiable" when no such instantiation can be found. *Enumerative instantiation* [22] is an alternative to MBQI that focuses on finding instantiations over the current set of ground terms that are not entailed in the current context. This has similar properties to MBQI but is more tailored for unsatisfiable instances.

Quantifier instantiation strategies that target specific theories have also been proposed. *Counterexample-guided instantiation* [24] is complete for specific theories with quantifiers that admit quantifier elimination, such as linear arithmetic and bit vectors. *SyQI* [18] is a more general purpose strategy that uses syntax-guided synthesis for enumerating instantiations and is effective for theories that otherwise do not have well-established instantiation strategies.

MBQI. MBQI iteratively refines a candidate model M constructed from the quantifier-free part of the problem. As shown in Algorithm 4, the strategy replaces function symbols in the body ϕ of the quantified formula with their interpretation in M. If $\neg\, \phi^M$ is $\mathcal{T}$-satisfiable, this means that a model M' exists, and the strategy returns the substitution $\{y_1 \mapsto y_1^{M'}, \ldots, y_n \mapsto y_n^{M'}\}$, where $y_i^{M'}$ represents a term denoting the interpretation for the variable y_i in M'.

Example 5. Let $\mathsf{a} : \mathit{Int}$ and $\mathsf{p} : \mathit{Int} \to \mathit{Bool}$. Let F be the input formula

$$(\forall y.\ \mathsf{p}\ y) \wedge 0 < \mathsf{a} < 2 \wedge \neg\, \mathsf{p}\ \mathsf{a}$$

where $y : \mathit{Int}$. In the SMT loop, our set of literals L is partitioned into $E = \{0 < \mathsf{a} < 2, \neg\, \mathsf{p}\ \mathsf{a}\}$, $Q_\mathrm{p} = \{\forall y.\ \mathsf{p}\ y\}$, and $Q_\mathrm{n} = \emptyset$. MBQI builds a model M from E—assume $\mathsf{a}^M = 1$ and $\mathsf{p}^M = \lambda x.\ x \neq 1$. (We abuse notation by denoting values using λ-terms.) It then considers the negation of the body of the quantified

Algorithm 4 The MBQI strategy

```
1: function INSTS_MBQI(q, E)
2:     assume q is ∀y_1, ..., y_n. φ
3:     let M be a model of E
4:     if ¬ φ^M is unsatisfiable then
5:         return ∅
6:     let M' be a model of ¬ φ^M
7:     return {{y_1 ↦ y_1^M', ..., y_n ↦ y_n^M'}}
```

formula in Q_{p} under the interpretation M, which is $\neg\,(\lambda x.\ x \neq 1)\ y$, which simplifies to $y = 1$ and has a model M' where $y^{M'} = 1$. Thus, MBQI generates the substitution $\{y \mapsto 1\}$, from which the instantiation lemma $(\forall y.\mathsf{p}\ y) \Longrightarrow \mathsf{p}\ 1$ is constructed and added to F. Now, the quantifier-free solver finds a contradiction.

Example 6. Let $\mathsf{f} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$. Let F be the higher-order formula

$$\forall x, y.\ y\ x \neq y\ (\mathsf{f}\ x)$$

where $x : \mathit{Int}$ and $y : \mathit{Int} \to \mathit{Int}$. The set L is partitioned into sets $E = \emptyset$, $Q_{\mathrm{p}} = \{\forall x, y.\ y\ x \neq y\ (\mathsf{f}\ x)\}$, and $Q_{\mathrm{n}} = \emptyset$. MBQI constructs a model M for E such that $\mathsf{f}^M = \lambda z.\ 0$. It then considers the negation of the body of the quantified formula in Q_{p} under the interpretation M. This is $\neg\, y\ x \neq y\ ((\lambda z.\ 0)\ x)$, which simplifies to $y\ x = y\ 0$ and has a model M' where $x^{M'} = 0$ and $y^{M'} = \lambda z.\ 0$. Thus, it generates the substitution $\{x \mapsto 0, y \mapsto \lambda z.\ 0\}$ based on the candidate model. Now, the quantifier-free solver finds a contradiction. Indeed, if we instantiate x and y with this substitution and β-reduce, we obtain $0 \neq 0$.

SyQI. SyQI uses SyGuS to choose instantiation terms. It aims to synthesize a term t for a variable x in a given formula $\forall x.\ \mathsf{p}\ x$ such that $\neg\,\mathsf{p}\ t$ holds. Each quantified variable is associated with a SyGuS grammar. The main advantage of SyQI is that, unlike MBQI, it does not require theory-specific quantifier instantiation procedures. The only parts that depend on the theory are the grammar and the $\mathcal{T}$-satisfiability check for the generated instances.

3 The Method

The core idea behind our new instantiation strategy, MBQI-Enum, is to integrate a SyGuS enumerator within MBQI, thereby enabling the generation of a broader set of candidate instantiations for quantified variables.

Instantiation Strategy. Instead of restricting instantiations to ground terms derived from the current MBQI model, our strategy uses a SyGuS grammar to produce additional candidate instantiations. This grammar is not limited to ground terms with the types of the quantified variables; rather, it incorporates

uninterpreted symbols gathered from the entire formula. As a result, it generates a more extensive and comprehensive language of terms.

For each quantified variable in a formula, our strategy performs iterative term enumeration. It generates candidate substitutions from the extended grammar and tests each enumerated term within the formula. For each enumerated term, the strategy tries to apply it as an instantiation for the quantified variable. For higher-order problems, it also considers λ-abstractions as candidate instantiations. If the strategy produces a useless instance according to the current model, it continues to the next candidate until a suitable instantiation is found or all possibilities are exhausted.

When none of the candidate instantiations derived from the SyGuS enumeration prove successful, MBQI-Enum reverts to the original MBQI model-derived instantiation. This fallback mechanism ensures that our strategy can in principle solve any problem that MBQI can solve.

Our initial motivation for developing MBQI-Enum was to increase cvc5's success rate on higher-order problems. Nevertheless, incorporating uninterpreted symbols gathered from the entire formula extends our approach's applicability to first-order problems using various SMT theories.

Our approach is presented in Algorithm 7. The strategy starts by invoking MBQI to generate a set of initial substitutions Σ, since the goal is to postprocess the substitutions generated by MBQI using a SyGuS enumerator. If Σ is empty, the strategy immediately returns an empty set, indicating that no valid instantiation could be found. Otherwise, it proceeds by initializing Σ to contain a single substitution σ. Next, it generates additional substitutions by extending the current one using the SyGuS enumerator.

Our strategy then iterates over the quantified variables y_i in the formula q to instantiate. For each variable, it constructs a grammar G_i used to guide the enumeration of candidate terms for substituting y_i. The enumeration starts by generating terms from G_i in a sequential manner. For each term e, the strategy creates a new substitution σ' by mapping y_i to e in the current substitution σ. It then checks whether the negation of the body ϕ under σ' is $\mathcal{T}$-satisfiable. This check serves to maintain the invariant that the negation of the body of the quantified formula, after applying the current substitution, remains $\mathcal{T}$-satisfiable. In other words, we want to ensure that the generated instantiation refutes the current model (cf. line 6 of Algorithm 4). If it does, σ is updated to σ', and the strategy moves on to the next variable. Once all variables have been considered, the strategy returns the refined substitution σ.

Choice of Grammar. Term enumeration is based on a SyGuS grammar. Choosing an appropriate grammar for each quantified variable is crucial for selecting the correct instantiations. Our strategy builds a set S of symbols based on three Boolean options: *syms_global*, *ext_vars*, and *syms_local*. These options specify which symbols from the formula F will be included when constructing the SyGuS grammar.

If no options are enabled, the set S is empty. If *syms_global* is enabled, all function symbols from the entire formula F are contained in S. If *ext_vars* is

Algorithm 7 The MBQI-Enum strategy

```
1: function INSTS_MBQI_FAST_SYGUS(q, E)
2:     assume q is ∀y_1, ..., y_n. φ
3:     let Σ ← INSTS_MBQI(q, E)
4:     if Σ = ∅ then
5:         return ∅
6:     else
7:         let Σ ← {σ}
8:     for each i ∈ {1, ..., n} do
9:         let G_i ← CHOOSE_GRAMMAR(q, y_i)
10:        for each j ∈ {1, 2, ...} do
11:            let e ← GET_ENUM_TERM(G_i, j)
12:            if e does not exist then
13:                break
14:            σ' ← σ[y_i ↦ e]
15:            if ¬ φσ' is sat then
16:                σ ← σ'
17:                break
18:    return σ
19: function CHOOSE_GRAMMAR(q, y_i)
20:    let F be the original input formula
21:    let S ← ∅
22:    if option_syms_global then
23:        S ← S ∪ symbols(F)
24:    if option_syms_local then
25:        S ← S ∪ symbols(q)
26:    if option_ext_vars then
27:        S ← S ∪ {y_{i+1}, ..., y_n}
28:    return grammar that generates terms of the same sort as y_i over symbols in S
```

enabled, S is augmented with bound variables from the formula q that are not yet instantiated. Finally, if *syms_local* is enabled, the set S also contains function symbols specifically from q. Based on these settings, our approach constructs a grammar that generates terms over the symbols in S, while ensuring that these terms are well-typed with respect to the type of y_i.

For higher-order variables, since we use η-long β-normal form, it is sufficient to consider only grammars that generate λ-abstractions. The variables bound by these λ-abstractions are considered terminal symbols of the grammar that generates the abstraction's body. For example, for a function variable whose arity is n and whose arguments are of the same base type as its return type, we add grammar rules of this form:

$$A ::= \lambda x_1, \ldots, x_n.\ B$$
$$B ::= x_1 \mid \cdots \mid x_n$$

Moreover, there will be additional rules for B and possibly for some of the grammar's other nonterminal symbols. This ensures that any λ instantiations are formed by enumerating the body B over the bound variables $x_1, \ldots, x_n$.

Example 8. Let $\mathsf{a} : \mathit{Int}$ and $\mathsf{p} : \mathit{Int} \to \mathit{Bool}$. Let F be the input formula

$$(\forall y.\, \neg\, \mathsf{p}\ (y\ \mathsf{a})) \wedge \mathsf{p}\ \mathsf{a}$$

where $y : \mathit{Int} \to \mathit{Int}$. The set L is partitioned into $E = \{\mathsf{p}\ \mathsf{a}\}$, $Q_\mathrm{p} = \{\forall y.\, \neg\, \mathsf{p}\ (y\ \mathsf{a})\}$, and $Q_\mathrm{n} = \emptyset$. MBQI generates substitutions such as $\{y \mapsto \lambda x.\, 0\}$, $\{y \mapsto \lambda x.\, 1\}$, ... As a result, the solver does not terminate. In contrast, MBQI-Enum augments these instantiations based on enumeration. On the first iteration of the SMT loop, MBQI-Enum considers the set $\Sigma = \{\{y \mapsto \lambda x.\, 0\}\}$ consisting of the first substitution generated by MBQI. MBQI-Enum first constructs a grammar for y. The set of symbols S is empty. The grammar is

$$\begin{aligned}
&\mathcal{A} ::= \lambda x.\, \mathcal{B}\\
&\mathcal{B} ::= x \mid 0 \mid 1 \mid \mathcal{B} + \mathcal{B} \mid \mathcal{B} - \mathcal{B} \mid \mathrm{ite}(\mathcal{B}, \mathcal{B}, \mathcal{B})\\
&\mathcal{C} ::= \mathsf{true} \mid \mathsf{false} \mid \mathcal{B} = \mathcal{B} \mid \mathcal{B} \leq \mathcal{B} \mid \neg \mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C}
\end{aligned}$$

Next, MBQI-Enum enumerates terms derived from the grammar and creates the substitution $\sigma' = \{y \mapsto \lambda x.\, x\}$ by updating σ with the enumerated term $\lambda x.\, x$ from the grammar. The strategy then checks whether the negation of the body of the quantified formula after applying σ' is $\mathcal{T}$-satisfiable. Indeed, if we instantiate y with $\lambda x.\, x$ and β-reduce, we obtain $\mathsf{p}\ \mathsf{a}$, which is $\mathcal{T}$-satisfiable. The substitution σ is then updated to $\{y \mapsto \lambda x.\, x\}$ and returned. Back in the SMT loop, the instantiation lemma $(\forall y.\, \neg\, \mathsf{p}\ (y\ \mathsf{a})) \Longrightarrow \neg\, \mathsf{p}\ \mathsf{a}$ is added to F. Now, the quantifier-free solver finds a contradiction.

The candidate substitutions generated by MBQI-Enum (M) are listed below:

Iter.	$q \in Q_\mathrm{p}$	E	$\mathrm{M}(q, E)$	New E
1	$\forall y.\, \neg\, \mathsf{p}\ (y\ \mathsf{a})$	$\{\mathsf{p}\ \mathsf{a}\}$	$\{\{y \mapsto \lambda x.\, x\}\}$	$\{\mathsf{p}\ \mathsf{a}, \neg\, \mathsf{p}\ \mathsf{a}\}$

The first column shows the number of the SMT loop iteration. The second column shows the quantified formula q, and the third column shows the set E of ground literals before every iteration. The fourth column shows the possible selection of substitutions of y that are considered with MBQI-Enum, and the fifth column shows the set E after every iteration.

In this example, MBQI-Enum was able to terminate in the first iteration, since it found a substitution for y that immediately leads to a refutation, whereas MBQI considers a repeating pattern of instantiations that leads to a timeout.

Another useful candidate substitution would have been $\{y \mapsto \lambda x.\, \mathsf{a}\}$. MBQI-Enum would have found this substitution as well if it had not terminated after finding $\{y \mapsto \lambda x.\, x\}$.

In an informal, preliminary evaluation on higher-order TPTP benchmarks, we determined that the most successful configuration enables *ext_vars* and

syms_local and leaves *syms_global* disabled. The following examples are based on this configuration.

Example 9. Let $\mathsf{p} : \mathit{Int} \to \mathit{Bool}$ and $\mathsf{f} : \mathit{Int} \to \mathit{Int}$. Let F be the input formula

$$\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)$$

where $y : \mathit{Int} \to \mathit{Int}$ and $z : \mathit{Int}$. The set L is partitioned into sets $E = \emptyset$, $Q_\mathrm{p} = \{\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)\}$, and $Q_\mathrm{n} = \emptyset$. MBQI generates substitutions such as $\{y \mapsto \lambda x.\, 0\}, \{y \mapsto \lambda x.\, 1\}, \ldots$, and the solver goes on forever. In contrast, MBQI-Enum adds the first substitution generated by MBQI, $\{y \mapsto \lambda x.\, 0\}$, to the set Σ and proceeds to postprocess it. Our strategy first constructs a grammar for y using the function symbols from q. The set of symbols used is $S = \{\mathsf{f}, \mathsf{p}\}$. The grammar is

$$\begin{aligned}
&\mathcal{A} ::= \lambda x.\, \mathcal{B}\\
&\mathcal{B} ::= x \mid \mathsf{f}\ \mathcal{B} \mid 0 \mid 1 \mid \mathcal{B} + \mathcal{B} \mid \mathcal{B} - \mathcal{B} \mid \mathrm{ite}(\mathcal{C}, \mathcal{B}, \mathcal{B})\\
&\mathcal{C} ::= \mathsf{true} \mid \mathsf{false} \mid \mathcal{B} = \mathcal{B} \mid \mathcal{B} \leq \mathcal{B} \mid \mathsf{p}\ \mathcal{B} \mid \neg \mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C}
\end{aligned}$$

In the first iteration, MBQI-Enum generates the substitution $\sigma = \{y \mapsto \lambda x.\, x\}$. The instantiation lemma $(\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)) \Longrightarrow \neg\, \forall z.\, \neg\, \mathsf{p}\ z \vee \mathsf{p}\ (\mathsf{f}\ z)$ is added to F. The set L is now partitioned into $E = \emptyset$, $Q_\mathrm{p} = \{\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)\}$, and $Q_\mathrm{n} = \{\neg\, \forall z.\, \neg\, \mathsf{p}\ z \vee \mathsf{p}\ (\mathsf{f}\ z)\}$. Next, the quantifier in Q_n is Skolemized. The Skolemization lemma $(\neg\, \forall z.\, \neg\, \mathsf{p}\ z \vee \mathsf{p}\ (\mathsf{f}\ z)) \Longrightarrow \mathsf{p}\ \mathsf{sk}_1 \wedge \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_1)$ is added to F. As a result, the set E is updated to $\{\mathsf{p}\ \mathsf{sk}_1, \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_1)\}$.

In the next iteration, the substitution σ is modified to $\{y \mapsto \lambda x.\, \mathsf{f}\ x\}$, incorporating the enumerated term $\lambda x.\, \mathsf{f}\ x$ from the grammar. The instantiation lemma $(\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)) \Longrightarrow \neg\, \forall z.\, \neg\, \mathsf{p}\ (\mathsf{f}\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)$ is then added to F. After Skolemization, the set E is augmented with $\{\mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2), \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2)\}$. The quantifier-free solver finds a contradiction. Indeed, if we instantiate y with $\lambda x.\, \mathsf{f}\ x$ in F, β-reduce, and Skolemize z, we obtain $\mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2) \wedge \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2)$.

The candidate substitutions generated by MBQI-Enum (M) are listed below:

Iter.	$q \in Q_\mathrm{p}$	E	M(q, E)	New E
1	$\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)$	$\emptyset$	$\{\{y \mapsto \lambda x.\, x\}\}$	$\{\mathsf{p}\ \mathsf{sk}_1, \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_1)\}$
2	$\forall y.\, \neg\, \forall z.\, \neg\, \mathsf{p}\ (y\ z) \vee \mathsf{p}\ (\mathsf{f}\ z)$	$\{\mathsf{p}\ \mathsf{sk}_1, \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_1)\}$	$\{\{y \mapsto \lambda x.\, \mathsf{f}\ x\}\}$	$\{\mathsf{p}\ \mathsf{sk}_1, \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_1)$ $\mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2), \neg\, \mathsf{p}\ (\mathsf{f}\ \mathsf{sk}_2)\}$

In this example, our strategy terminated in the second iteration, whereas MBQI would lead the solver to time out.

Example 10. In this example, our strategy is run with and without the *syms_global* option enabled. Let u be an uninterpreted sort, and let $\mathsf{a} : u$ and $\mathsf{b} : u$. Let F be the input formula

$$(\forall x, y, z.\, x\ y = x\ z) \wedge \mathsf{a} \neq \mathsf{b}$$

where $x : (u \to u) \to u$, $y : u \to u$, and $z : u \to u$. The set L is partitioned into $E = \{\mathsf{a} \neq \mathsf{b}\}$, $Q_\mathrm{p} = \{\forall x, y, z.\, x\ y = x\ z\}$, and $Q_\mathrm{n} = \emptyset$. MBQI-Enum fails

to construct a grammar for a variable x, y, or z that has any terms of the same type as the variable. As a result, it cannot generate any substitutions, and the solver gives up early. In contrast, when the *syms_global* option is enabled in MBQI-Enum, function symbols from the entire formula F are used to construct a grammar for each variable x, y, and z. For these variables, the set of symbols is $\{\mathsf{a}, \mathsf{b}\}$. The grammar for x follows:

$$\begin{aligned}\mathcal{A} &::= \lambda w.\ \mathcal{B}\\ \mathcal{B} &::= w\ \mathcal{B} \mid \mathsf{a} \mid \mathsf{b} \mid \mathrm{ite}(\mathcal{C}, \mathcal{B}, \mathcal{B})\\ \mathcal{C} &::= \mathsf{true} \mid \mathsf{false} \mid \mathcal{B} = \mathcal{B} \mid \neg\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C}\end{aligned}$$

(Since w is of unary function type, we pass an argument corresponding to the nonterminal $\mathcal{B}$ in the second grammar rule.) The grammar for y and z follows:

$$\begin{aligned}\mathcal{A} &::= \lambda w.\ \mathcal{B}\\ \mathcal{B} &::= w \mid \mathsf{a} \mid \mathsf{b} \mid \mathrm{ite}(\mathcal{C}, \mathcal{B}, \mathcal{B})\\ \mathcal{C} &::= \mathsf{true} \mid \mathsf{false} \mid \mathcal{B} = \mathcal{B} \mid \neg\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C}\end{aligned}$$

Our strategy then enumerates terms derived from the grammar and builds the substitutions $\{x \mapsto \lambda w.\ w\ \mathsf{b}\}, \{y \mapsto \lambda w.\ w\}$, and $\{z \mapsto \lambda w.\ \mathsf{a}\}$. The instantiation lemma $(\forall x, y, z.\ x\ y = x\ z) \Longrightarrow \mathsf{b} = \mathsf{a}$ is added to F. Now, the quantifier-free solver finds a contradiction. Indeed, if we instantiate x with $\lambda w.\ w\ \mathsf{b}$, y with $\lambda w.\ w$, and z with $\lambda w.\ \mathsf{a}$ in F and β-reduce, we obtain $\mathsf{a} = \mathsf{b} \wedge \mathsf{a} \neq \mathsf{b}$.

The candidate substitutions generated by MBQI-Enum (M) are listed below:

Iter.	$q \in Q_\mathrm{p}$	E	$\mathrm{M}(q, E)$	New E
1	$\forall x, y, z.\ x\ y = x\ z$	$\{\mathsf{a} \neq \mathsf{b}\}$	$\emptyset$	$\{\mathsf{a} \neq \mathsf{b}\}$

The same information is provided for MBQI-Enum with the option *syms_global* enabled (M+g) below:

Iter.	$q \in Q_\mathrm{p}$	E	$(\mathrm{M{+}g})(q, E)$	New E
1	$\forall x, y, z.\ x\ y = x\ z$	$\{\mathsf{a} \neq \mathsf{b}\}$	$\{\{x \mapsto \lambda w.\ w\ \mathsf{b}\},$ $\{y \mapsto \lambda w.\ w\},$ $\{z \mapsto \lambda w.\ \mathsf{a}\}\}$	$\{\mathsf{a} \neq \mathsf{b}, \mathsf{a} = \mathsf{b}\}$

In this example, MBQI-Enum with *syms_global* was able to terminate in the first iteration, since it found a substitution for the quantified variables that leads to a refutation. By contrast, default MBQI-Enum does not terminate since it cannot build any substitutions.

4 Implementation and Heuristics

We implemented MBQI-Enum as an extension of cvc5's implementation of MBQI. Our strategy is invoked after the current MBQI strategy returns a candidate instantiation (line 3 of Algorithm 7).

For each variable, our algorithm chooses a grammar (line 9) and initializes a term enumeration data structure. Since the choice of the grammar is fixed over the course of solving, the grammar is constructed only once. Our implementation uses the utility for fast SyGuS enumeration described by Reynolds et al. [23] as a black box. Since the grammar for each variable is fixed, we can cache the enumeration and invoke this utility only on line 11 of Algorithm 7, when j is larger than the number of terms we have generated on a previous run, where we notice that a term that was skipped in a previous call to this method may be incorporated into instantiations on later calls.

On line 15 of Algorithm 7, we use cvc5's ability to call a copy of itself as a subsolver. As an optimization, this satisfiability check can be avoided if the query to check simplifies to "true" or "false."

5 Evaluation

We extensively evaluated our cvc5 implementation of MBQI-Enum on both higher- and first-order benchmarks.

Setup. As the base configuration, we use the setup that we found to be the most successful in a preliminary evaluation: MBQI-Enum with the options *ext_vars* and *syms_local* enabled by default. We denote this configuration by cvc5[M].

We first compare the performance of the base configuration against traditional instantiation techniques: cvc5[e], which uses enumerative instantiation [22]; cvc5[s], which uses SyQI [18]; cvc5[c], which uses counterexample-guided instantiation [24]; and cvc5[m], which uses MBQI [9].

Additionally, for higher-order problems, we also include a comparison with the state-of-the-art provers Vampire [6] and Zipperposition [28]. For Vampire, we used its portfolio mode, while Zipperposition was run in its so-called "best" mode, since it does not include a portfolio.

Next, we compare the base configuration on first-order benchmarks with three state-of-the-art SMT solvers: Z3 [15], the only SMT solver besides cvc5 that supports all the logics handled by our implementation; Bitwuzla [16], which supports only logics without arithmetic; and Boolector [17], which implements the most closely related approach to ours, counterexample-guided model synthesis [21], but focuses only on the theory of bit vectors.

Finally, we compare the performance of all four MBQI-Enum configurations on both higher-order and first-order problems. In this evaluation, we toggled one option at a time: cvc5[M−x] denotes MBQI-Enum with *ext_vars* disabled, cvc5[M−ℓ] denotes MBQI-Enum with *syms_local* disabled, and cvc5[M+g] denotes MBQI-Enum with *syms_global* enabled.

We performed all experiments on a system with a 40-core Intel Xeon Silver 4114 processor at 2.20 GHz and with 192 GB of RAM using Debian Bookworm as the operating system. We used a time limit of 60 seconds for each benchmark.

Table 1. MBQI-Enum vs. other strategies and provers on TPTP TH0 benchmarks

	Vampire	Zipperposition	cvc5[e]	cvc5[s]	cvc5[m]	cvc5[M]
Satisfiable	6	0	72	78	121	**129**
Unsatisfiable	**1757**	1499	1643	1304	1637	1670
Total	1763	1499	1715	1382	1758	**1799**
Unknown	0	0	350	38	127	59
Timeouts	999	1263	697	1342	877	904

Table 2. MBQI-Enum configurations on TPTP TH0 benchmarks

	cvc5[M]	cvc5[M−x]	cvc5[M−ℓ]	cvc5[M+g]
Satisfiable	**129**	**129**	122	**129**
Unsatisfiable	1670	1665	1655	**1672**
Total	1799	1794	1777	**1801**
Unknown	59	65	88	56
Timeouts	904	903	897	905

Higher-Order Problems. The higher-order part of the experiments was carried out on monomorphic higher-order problems (TH0) from version 9.0.0 of the TPTP library [27]. The benchmark set consists of 2762 problems. From the 3962 TH0 problems, we excluded 1200 benchmarks that one or more systems could not parse (e.g., because they use arithmetic).

The results are summarized in Table 1. In this and the following tables, bold indicates the most successful system. Notably, our approach achieves the highest total count of solved benchmarks, surpassing the nearest competitor by 36 solved problems. Overall, it outperforms all other cvc5 strategies as well as Zipperposition in higher-order logic and solves 87 fewer unsatisfiable problems than Vampire. Our strategy's advantage over Zipperposition likely stems from using Zipperposition's "best" mode instead of a portfolio. Remarkably, our strategy manages to solve 129 satisfiable problems, whereas Vampire solves only 6, and Zipperposition none.

Although our approach is based on both MBQI and SyQI, it is considerably stronger than either of those techniques used individually. Specifically, compared with MBQI, which was until now the most successful strategy in cvc5, MBQI-Enum solves an additional 41 problems without any losses. When compared with SyQI, our strategy solves 417 more benchmarks while also incurring no losses.

Table 2 shows the evaluation of the different configurations of MBQI-Enum on higher-order problems. We see that all three options are beneficial, but for *syms_global* the difference is only two problems. (In our preliminary evaluation, we had found *syms_global* to be slightly harmful, which is why we disabled it by default.)

First-Order Problems. The experiments on first-order problems were conducted on the SMT-LIB benchmarks [5] from April 2024, focusing on logics that support quantifiers. We included logics involving theories such as floating-point arithmetic, linear and nonlinear arithmetic, and bit vectors. Overall, we consider the logics BV (bit vectors), FP (floating-point arithmetic), LIA (linear integer arithmetic), LRA (linear real arithmetic), NIA (nonlinear integer arithmetic), NRA (nonlinear real arithmetic), and their combinations: BVFP, BVFPLRA, and FPLRA. We also incorporate ABV (arrays and bit vectors) and UFBV (uninterpreted functions with bit vectors). In total, our benchmark set consists of 21 605 problems.

Table 3. MBQI-Enum vs. other techniques and solvers on SMT-LIB benchmarks

Library	Boolector		Bitwuzla		Z3		cvc5[e]		cvc5[s]		cvc5[c]		cvc5[m]		cvc5[M]	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
BV	585	4980	**642**	5066	547	5001	202	4836	313	4911	437	**5190**	611	4835	594	5076
SAT+UNSAT	5565		**5708**		5548		5038		5224		5627		5446		5670	
ABV			378	47	387	103	21	**930**	618	237	17	90	**790**	135	731	157
ABVFP			24	0	29	2	10	**3**	15	0	13	0	**31**	0	**31**	0
ABVFPLRA			32	1	**55**	**3**	14	2	18	1	18	2	41	2	42	2
BVFP			**179**	**12**	164	5	26	**12**	102	3	106	0	167	4	169	7
BVFPLRA			**233**	**25**	205	18	80	**25**	119	24	111	24	220	24	226	**25**
FP			**130**	2176	17	1802	113	1591	99	2040	116	2015	115	2059	116	**2223**
FPLRA			**37**	0	23	0	20	0	24	0	22	0	**37**	0	36	0
UFBV			26	**120**	**43**	103	8	103	9	108	8	52	23	84	21	105
Subtotal			1681	7447	1470	7037	494	7502	1317	7324	848	7373	**2035**	7143	1966	**7595**
SAT+UNSAT			9128		8507		7996		8641		8221		9178		**9561**	
LIA					140	230	12	170	**150**	236	**150**	**266**	**150**	167	149	239
LRA					**760**	**1361**	468	1117	478	1130	593	1303	545	1123	557	1161
NIA					65	**144**	16	43	49	45	64	**144**	67	47	**80**	61
NRA					**3**	**3806**	1	3802	1	3785	**3**	3801	**3**	3712	**3**	3802
Total					2438	12 578	991	12 634	1995	12 520	1658	**12 887**	**2800**	12 192	2755	12 858
SAT+UNSAT					15 016		13 625		14 515		14 545		14 992		**15 613**	

The results, summarized in Table 3, show that our approach performs remarkably well against all other cvc5 configurations, as well as against Boolector, Bitwuzla, and Z3, across all SMT logics. Notably, it achieves the highest total count of benchmarks solved, surpassing the nearest competitor by 597 solved problems. Our strategy solves the most satisfiable problems in ABVFP and NIA and achieves the highest number of unsatisfiable benchmarks solved in FP.

For the evaluated SMT theories, our strategy is a clear improvement over previous instantiation strategies. When compared with MBQI, it successfully solves an additional 701 problems while incurring a loss of 79 problems across all logics. The raw evaluation data also reveals a notable reduction in timeouts, decreasing from 3491 to 2343.

Our strategy also substantially outperforms enumerative instantiation and SyQI across most benchmark categories. Both of these strategies share an enumerative nature. The former relies on evolving ground terms within the current context, while the latter employs a fixed grammar derived from the initial set of terms. Overall, enumerative instantiation and SyQI perform clearly better than

Table 4. MBQI-Enum configurations in SMT-LIB benchmarks

Library	cvc5[M]		cvc5[M−x]		cvc5[M−ℓ]		cvc5[M+g]	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
ABV	731	157	731	157	647	135	**737**	**159**
ABVFP	31	0	31	0	31	0	**32**	0
ABVFPLRA	**42**	2	**42**	2	41	2	38	2
BV	594	5076	594	5075	**616**	4843	600	**5087**
BVFP	169	**7**	169	**7**	167	4	**176**	2
BVFPLRA	226	**25**	226	**25**	220	24	**231**	**25**
FP	**116**	2223	**116**	2223	115	2092	**116**	**2223**
FPLRA	36	0	36	0	**37**	0	36	0
LIA	149	**239**	149	**239**	**150**	167	149	**239**
LRA	**557**	**1157**	**557**	**1157**	546	1129	**557**	**1157**
NIA	**80**	**61**	**80**	**61**	68	47	78	56
NRA	3	**3802**	3	**3802**	3	3712	3	**3802**
UFBV	21	**105**	21	**105**	**24**	89	17	99
Total	2755	**12 858**	2755	12 857	2665	12 244	**2771**	12 854

MBQI on unsatisfiable benchmarks (+442), but they underperform on satisfiable benchmarks (−805). This highlights the need for a hybrid approach that combines model-based and enumerative techniques. Our strategy incorporates the enumerative aspects of SyQI while enhancing the model-based features of MBQI to generate instantiations. This is likely why it outperforms all the mentioned configurations. Our strategy also matches or outperforms counterexample-guided instantiation in most logics. However, in logics such as LRA, counterexample-guided instantiation is expected to perform better due to its specialized handling of such theories.

Compared with Boolector, our strategy outperforms it on both satisfiable and unsatisfiable benchmarks, solving 105 more benchmarks overall. Compared with Bitwuzla and Z3, our approach performs very well across various logics, often closely matching or even surpassing both competitors. Overall, our strategy solves 433 more benchmarks than Bitwuzla and 597 more than Z3 in total. Bitwuzla is generally stronger for satisfiable problems, which is not surprising because MBQI-Enum is primarily designed for deriving contradictions. Z3's higher success rate for real arithmetic is likely attributable to its well-established instantiation strategies for these theories.

Finally, the evaluation of the various configurations of MBQI-Enum on first-order SMT-LIB benchmarks across different theories is shown in Table 4. We see that most configurations perform similarly; however, MBQI-Enum without the *syms_local* option enabled shows significantly poorer performance.

In summary, our approach is highly effective on first-order SMT-LIB benchmarks, solving the highest number of benchmarks. With refinements tailored to specific logics, we suspect that its performance could be improved further.

6 Related Work

Mainstream approaches for quantifier instantiation in SMT are typically centered around E-matching [14]. Conflict-based instantiation [3,11,26] can improve the solver's ability to answer "$\mathcal{T}$-unsatisfiable" by prioritizing instantiations that induce quantifier-free conflicts. As a whole, these techniques are generally incomplete and do not target specific background theories. For satisfiable instances, Ge and de Moura [9] introduced MBQI, which is complete for certain fragments. Finite model finding [25] is a variant of this technique that targets quantified formulas whose domains are small and finite. Approaches for quantified formulas in higher-order logic are discussed by Barbosa et al. [4], but, in contrast to this work, they are based on (higher-order) E-matching.

Other approaches for higher-order logic, notably in Vampire [6] and Zipperposition [28], rely on superposition. Vampire has been initially extended to handle higher-order reasoning using applicative first-order logic with combinators. Since this proved insufficient for problems requiring complex unifiers, its superposition calculus was later enhanced with native λ-abstractions and a depth-bounded version of higher-order unification [6]. As for Zipperposition, it also uses a superposition calculus that directly supports higher-order terms. It tackles the challenge of higher-order unification by using techniques such as pattern unification and heuristics to manage undecidability issues.

Certain background theories admit quantifier elimination, which can be handled using domain-specific instantiation strategies. Specifically, efficient and complete instantiation procedures have been developed for quantified linear arithmetic [24] and quantified bit vectors [19]. These techniques require specific knowledge of the background theory.

Other recent works on quantifier instantiation have pursued enumeration as a pragmatic means for discovering useful instantiations. Reynolds et al. [22] introduced enumerative instantiation as an alternative to MBQI, which primarily focused on first-order logic in the empty theory. This technique has been further studied in more recent works, where more advanced selection strategies are used for instantiations, including those based on machine learning [12,13,20].

The closest related works to ours are counterexample-guided model synthesis [21] and SyQI [18], which both focus on enumerative approaches for finding useful instantiations in rich background logics. The former was implemented in the Boolector [17] solver; it was limited to selected first-order theories and did not handle higher-order logic. The latter work can potentially be used for any theory but does not leverage MBQI for guiding the instantiation procedure. Our evaluation shows that our MBQI-Enum strategy generally outperforms SyQI overall.

7 Conclusion

We presented a new strategy, MBQI-Enum, for instantiating quantifiers in SMT solvers. It extends MBQI with the SyGuS enumerator, thereby augmenting the number of instantiations considered at every iteration. The main strength of our

strategy is that it combines the fast model-finding capabilities of MBQI and the diversity of terms considered by SyQI. MBQI generates very specific instances; by resorting to a grammar, the terms in our instantiations are more abstract and therefore tend to lead to more useful instances. We implemented the strategy in cvc5 and found that it helps solve many first- and higher-order problems from SMT-LIB and TPTP for which cvc5 previously either timed out or gave up early.

Several aspects of our approach present opportunities for future work. First, we could improve performance by enhancing the quantifier-free solver to better integrate with our instantiation approach. Moreover, although our instantiation technique is designed to be generic, we could tailor it to individual SMT logics. Finally, we could develop more sophisticated instantiation strategies for higher-order logic. By designing methods that can more intelligently navigate the space of enumerated terms, we should be able to improve the solver's ability to handle complex higher-order problems.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. We also thank Pascal Fontaine and Mark Summerfield, who provided many comments on earlier drafts.

This research was cofunded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Andrews, B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, vol. 27. Springer, 2nd edn. (2002)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fishman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442 (2022)
3. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part II. LNCS, vol. 10206, pp. 214–230. Springer (2017)
4. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 2019. LNCS, vol. 11716, pp. 35–54. Springer (2019)
5. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016)
6. Bhayat, A., Suda, M.: A higher-order Vampire (short paper). In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) IJCAR 2024. LNCS, vol. 14739, pp. 75–85. Springer (2024)
7. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 151–156. Springer (2009)

8. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940)
9. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer (2009)
10. Gordon, M.J.C.: Introduction to the HOL system. In: Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J. (eds.) TPHOL 1991. pp. 2–3. IEEE (1991)
11. Hoenicke, J., Schindler, T.: Incremental search for conflict and unit instances of quantified formulas with E-matching. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 534–555. Springer (2021)
12. Jakubův, J., Janota, M., Piotrowski, B., Piepenbrock, J., Reynolds, A.: Selecting quantifiers for instantiation in SMT. In: Graham-Lengrand, S., Preiner, M. (eds.) SMT 2023. CEUR Workshop Proceedings, vol. 3429, pp. 71–77. CEUR-WS.org (2023)
13. Janota, M., Barbosa, H., Fontaine, P., Reynolds, A.: Fair and adventurous enumeration of quantifier instantiations. In: FMCAD 2021. pp. 256–260. IEEE (2021)
14. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 183–198. Springer (2007)
15. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
16. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) CAV 2023. LNCS, vol. 13965, pp. 3–17. Springer (2023)
17. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014)
18. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Syntax-guided quantifier instantiation. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12652, pp. 145–163. Springer (2021)
19. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part II. LNCS, vol. 10982, pp. 236–255. Springer (2018)
20. Piepenbrock, J., Janota, M., Urban, J., Jakubův, J.: First experiments with neural cvc5. In: Bjørner, N.S., Heule, M., Voronkov, A. (eds.) LPAR 2024. EPiC Series in Computing, vol. 100, pp. 264–277. EasyChair (2024)
21. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 264–280. Springer (2017)
22. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D. (ed.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer (2018)
23. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part II. LNCS, vol. 11562, pp. 74–83. Springer (2019)
24. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer (2015)
25. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 640–655. Springer (2013)
26. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: FMCAD 2014. pp. 195–202. IEEE (2014)
27. Sutcliffe, G.: The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)

28. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. Journal of Automated Reasoning **66**(4), 541–564 (2022)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.







Enumerating Choice Terms in Model-Based Quantifier Instantiation

Lydia Kondylidou[1], Andrew Reynolds[2], Jasmin Blanchette[1], and Cesare Tinelli[2]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
l.kondylidou@lmu.de, jasmin.blanchette@lmu.de
[2] The University of Iowa, Iowa City, United States
andrew-reynolds@uiowa.edu, cesare-tinelli@uiowa.edu

Abstract. Satisfiability modulo theories (SMT) solvers are widely used for determining the satisfiability of logical formulas with respect to background theories. SMT solvers are traditionally based on first-order logic, but some also support higher-order logic. Recently, Kondylidou et al. introduced model-based quantifier instantiation with fast enumeration (MBQI-Enum), a quantifier instantiation strategy that works for both logics. A weakness of MBQI-Enum is that it does not find refutations when Hilbert choice terms are necessary. In this work, we present an extension of MBQI-Enum that enables it to reason effectively about Hilbert's choice operator. The extended strategy substantially increases the success rate of the SMT solver cvc5 on higher-order benchmarks.

1 Introduction

Satisfiability modulo theories (SMT) solvers combine a Boolean satisfiability (SAT) solver with decision procedures for interpreted theories. They work by refutation: They assume the negation of the conjecture as an axiom and try to establish the unsatisfiability (i.e., provability) of the input problem. Several SMT solvers, including Bitwuzla [25], Boolector [26], CVC4 [5], cvc5 [3], veriT [8], and Z3 [23], support quantifiers via Skolemization and instantiation.

Most SMT solvers are based on first-order logic, but a few support higher-order logic. Specifically, CVC4, its successor cvc5, and a prototype version of veriT have been extended to parse and solve higher-order problems [4]. In principle, higher-order logic is often more convenient than first-order logic for expressing problems, particularly those involving binders (e.g., λx, $\sum_i$, $\prod_i$, $\int_x$).

A crucial aspect of higher-order SMT solvers is their use of quantifier instantiation strategies. Barbosa et al. [4] partly extended the first-order E-matching strategy [22] to a higher-order setting. Recently, Kondylidou et al. [19] introduced model-based quantifier i nstantiation w ith f ast e numeration (MBQI-Enum), a strategy that extends model-based quantifier i nstantiation (MBQI) [15] with syntax-guided synthesis (SyGuS) [27] techniques. While traditional MBQI relies only on ground terms from the MBQI model to instantiate quantified variables, MBQI-Enum broadens this approach by generating a wider range of candidate

© The Author(s) 2026
S. Junges and G. Katz (Eds.): TACAS 2026, LNCS 16505, pp. 295–314, 2026.
https://doi.org/10.1007/978-3-032-22752-2_15

instantiations. It does so by enumerating terms guided by a SyGuS grammar, which enables it to construct more complex instantiations involving, for instance, identity functions and uninterpreted symbols. MBQI-Enum was found to be the most successful strategy [19, Sect. 5] for solving higher-order problems from the TPTP [39] library.

A weakness of MBQI-Enum is that it does not attempt instantiations containing Hilbert's choice operator. A Hilbert choice expression has the form $\varepsilon x.\ \varphi$, where variable x is bound in formula φ. The entire expression denotes some value of x that satisfies φ, if such a value exists; otherwise, it denotes an arbitrary value from the type of x. Sometimes the ε operator occurs in the input problem, in which case the SMT solver must reason about it. Even if ε is absent, an ε term might be useful in quantifier instantiations. The ε operator is reputed to be difficult to reason about because it is characterized by a higher-order axiom.

In this paper, we extend MBQI-Enum so that it instantiates quantifiers with terms that include Hilbert's choice operator. This extension requires three main modifications to MBQI-Enum. First, we augment the SyGuS grammar to include ε terms. Second, we introduce fresh Skolem symbols that represent these ε terms, since this is simpler than implementing ε terms throughout the SMT solver. Finally, we generate lemmas corresponding to instances of ε's characteristic axiom and add them to the solver.

Example 1. We compare our extended MBQI-Enum with the original on TPTP benchmark `SEV431^1`. Let u be an uninterpreted sort, and let f be an uninterpreted symbol of type $u \to u$. Consider the second-order problem consisting of the injectivity axiom $\forall x, y.\ \mathsf{f}\ x = \mathsf{f}\ y \Longrightarrow x = y$ for f and the negated conjecture $\forall g.\ \exists z.\ g\ (\mathsf{f}\ z) \neq z$, where g has type $u \to u$ and x, y, z have type u. MBQI-Enum generates the instantiations $\lambda y.\ y$, $\lambda y.\ \mathsf{f}\ y$, $\lambda y.\ \mathsf{f}\ (\mathsf{f}\ y)$, ... for g. As a result, the SMT solver does not terminate. By contrast, our strategy instantiates g with the term $\lambda y.\ \varepsilon x.\ y = \mathsf{f}\ x$ using the grammar. This term denotes the left inverse of f. Then the strategy introduces a fresh Skolem symbol h that represents this term. The lemma $\forall y.\ \neg\,(\exists x.\ y = \mathsf{f}\ x) \vee y = \mathsf{f}\ (\mathsf{h}\ y)$ characterizing h is added to the problem. After instantiating g with h, the solver can derive a contradiction with the injectivity axiom. This implies that the unnegated conjecture $\exists g.\ \forall z.\ g\ (\mathsf{f}\ z) = z$ is entailed by the axiom. ■

We implemented the extended MBQI-Enum strategy in the cvc5 solver. We evaluated it on the higher-order benchmarks of the TPTP library [39]. The results show that Hilbert's choice is often useful for establishing the unsatisfiability of an input problem without introducing much overhead in problems where it is not needed. Remarkably, our extension solves a strict superset of the problems solved by the original MBQI-Enum.

We also compare our approach with other cvc5 strategies and find that it outperforms them. Finally, we compare cvc5 equipped with our strategy with the provers Satallax [10], Vampire [7], and Zipperposition [40], representing the state of the art in higher-order automated theorem proving, and find that it is highly competitive. The raw evaluation data are available online [21]. Our source

code, along with instructions for reproducing the experiments, is also available online [20].

2 Preliminaries

Our approach builds on higher-order logic with Hilbert's choice, SMT with quantifiers, and MBQI-Enum. Below, we briefly introduce these concepts.

Higher-Order Logic with Hilbert's Choice. Monomorphic higher-order logic [1, 16], also called simple type theory [11], generalizes classical first-order logic by allowing quantification over functions. The syntax distinguishes between types and terms. Types τ are either base types, or sorts, κ or applications of the function type constructor $\to$ to two types: $\tau_1 \to \tau_2$. The type of Booleans is denoted by o.

The term language is based on the simply typed λ-calculus, where terms $t, e, \ldots$ are inductively defined as variables $g, x, y, z, \ldots$, symbols $\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{h}, \mathsf{p}, \mathsf{r} \ldots$ (possibly of function type), term applications $t\ t'$, and λ-abstractions $\lambda x.\ t$, where x is the bound variable and t is the abstraction's body. A variable is free in a term if it is not bound by an enclosing λ-abstraction. Terms are syntactically equal modulo α-, β-, and η-conversion, meaning, for example, that $(\lambda x.\ \mathsf{f}\ x\ x)\ \mathsf{c}$ is syntactically equal to $\mathsf{f}\ \mathsf{c}\ \mathsf{c}$. Terms of type $\tau_1 \to \cdots \to \tau_n \to o$ are called predicates. A term of type o is called a formula.

A common extension of higher-order logic is *Hilbert's choice operator* ε [17,18, 37]: If x is a variable of type τ and φ is a formula that contains x, then $\varepsilon\ (\lambda x.\ \varphi)$, abbreviated as $\varepsilon x.\ \varphi$, is a term of type τ. Intuitively, the term $\varepsilon x.\ \varphi$ denotes *some* element x of type τ for which the formula φ holds if such an element exists. If no such element exists, it denotes an arbitrary element of type τ. Unlike existential quantification, which only asserts the existence of such an element, the choice operator yields the element. Hilbert's choice operator is characterized by the higher-order axiom $\forall y.\ (\exists x.\ y\ x) \Longrightarrow y\ (\varepsilon x.\ y\ x)$. The operator makes it possible to denote functions that exist according to the standard semantics of higher-order logic but that otherwise cannot be expressed (cf. Example 1). Some treatments of Hilbert's choice include an axiom $\forall y, z.\ y = z \Longrightarrow (\varepsilon x.\ y\ x) = (\varepsilon x.\ z\ x)$, but this is a special case of congruence, which is built into higher-order logic.

Quantifier Instantiation in SMT. To reason about the satisfiability of quantified formulas, SMT solvers typically rely on a combination of Skolemization and heuristic instantiation: Universal quantifiers occurring positively in a formula are instantiated, while those occurring negatively are Skolemized. Dually, existential quantifiers occurring positively are Skolemized, whereas those occurring negatively are instantiated.

Let $\mathcal{T}$ be a theory, or combination of theories, over a set of interpreted symbols, and let F be a (conjunctively understood) set of formulas, or *problem*, over the signature of $\mathcal{T}$. To determine the satisfiability of F in $\mathcal{T}$, typical SMT solvers rely on a *quantifier-free subsolver* for reasoning about quantifier-free formulas and an *instantiation module* for reasoning about quantified formulas. The

former, which combines a SAT solver and a specialized solver for the theory $\mathcal{T}$, enumerates potential models of an abstraction of F in which quantified subformulas are treated as propositional variables. The SAT solver proposes candidate models for F at the propositional level—truth assignments to atomic formulas in F that propositionally satisfy F—while the theory solvers check these models for theory-specific conflicts. If all candidate models generate conflicts, F is declared unsatisfiable. Otherwise, the instantiation module handles the quantified subformulas in F by generating selected instances of them and adding them to F. The added instances preserve the satisfiability of F. Instances with quantifiers will be possibly instantiated further, whereas those with no quantifiers will be directly processed by the quantified-free subsolver.

Instantiation is performed following user-selectable strategies that compute substitutions σ for the top-level variables in quantified formulas of F, mapping those variables to ground (i.e., variable-free) terms. More precisely, for selected quantified formulas q of the form $\forall x_1, \ldots, x_n.\ \psi$ in F and selected substitutions σ that map each variable x_i to a ground term t_i, the instantiation module produces *lemmas* of the form $q \Longrightarrow \psi\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ that are then added to the set F. Existentially quantified formulas in F are instantiated by Skolemization: for each selected formula q of the form $\exists x_1, \ldots, x_n.\ \psi$ in F, n fresh Skolem symbols sk_1, ..., sk_n are introduced and a corresponding lemma of the form $q \Longrightarrow \psi\{x_1 \mapsto \mathsf{sk}_1, \ldots, x_n \mapsto \mathsf{sk}_n\}$ is added to F. The main SMT loop is then reentered with the new F.

This iterative process continues until the SMT solver finds a model for F with respect to $\mathcal{T}$ (which is itself a difficult task given the presence in F of quantified formulas [14,15,32,34]) or the quantifier-free subsolver determines F to be unsatisfiable in $\mathcal{T}$. Of course, at any point, the SMT solver may run out of time or memory.

To generate relevant ground terms for instantiating universal quantifiers, SMT solvers rely on *instantiation strategies*. For first-order logic with theories, some strategies are refutationally complete and enable SMT solvers to offer semidecision procedures: They will find a proof of unsatisfiability if one exists, but they may not terminate otherwise. Higher-order logic—which allows terms to include functions as arguments, λ-abstractions, and partial applications—remains semidecidable under Henkin semantics but is more challenging to support in practice than first-order logic. The strategies implemented in CVC4, cvc5, and veriT are pragmatic and do not seek refutational completeness [4, 19].

For higher-order logic, cvc5 initially supported only an extension of the E-matching [22] instantiation strategy. New strategies that also partly work on higher-order problems were later added to the solver. Specifically, MBQI [15] is a strategy that iteratively refines a candidate model constructed from the quantifier-free part of the problem and then attempts to refute the model by generating ground instances of the quantified formulas that may be falsified by the model. It has been used on higher-order benchmarks and has proved somewhat effective on these [19, Sect. 5]. Common techniques such as conflict-guided instantiation [35], enumerative instantiation [30], and counterexample-

guided instantiation [33] can also be used on higher-order problems, but they have generally proved less effective [19, Sect. 5], which is unsurprising given that they were originally designed for first-order reasoning. SyQI [27], a syntax-guided synthesis approach, works well on some higher-order problems since it can generate terms by synthesizing candidate expressions with the help of a grammar [19, Sect. 5].

MBQI-Enum. MBQI-Enum [19] is a recent addition to cvc5 that integrates a SyGuS-based enumerator directly into MBQI. For each quantified variable in the formula, MBQI-Enum first constructs a grammar. It then iteratively enumerates candidate terms generated from the grammar and checks each resulting instance against the current model. If one instance refutes the current model, the strategy builds a substitution by mapping the quantified variable to the current enumerated term. If the instance fails to refute the model, the enumerator proceeds to the next candidate term. If all candidates are exhausted without success, the strategy reverts to standard MBQI. This fallback mechanism guarantees that MBQI-Enum can, in principle, solve any problem that MBQI can solve.

A key aspect of MBQI-Enum is the construction of the SyGuS grammar that guides the enumeration. The grammar is created from a set of symbols including uninterpreted symbols extracted from the entire formula and bound variables that have not yet been instantiated. For higher-order variables, the grammar incorporates rules for generating λ-abstractions. The enumerator then systematically constructs λ-expressions by generating terms over the abstractions' argument variables.

Example 2. We consider a modified version of the TPTP benchmark `SYO288^5`, which is over the theory $\mathcal{T}$ of equality with uninterpreted sorts and functions. Let u be an uninterpreted sort. Let f be a function symbol $\mathsf{f} : u \to u$ and consider the second-order conjecture $\exists y.\ \forall z.\ y\ z = \mathsf{f}\ z$, where y is a variable of type $u \to u$ and z is a variable of type u. To prove its validity in $\mathcal{T}$, we ask the SMT solver to prove the unsatisfiability of its negation, or, equivalently, of the input problem:

$$F = \{\forall y.\ \exists z.\ y\ z \neq \mathsf{f}\ z\}$$

MBQI-Enum first constructs a SyGuS grammar for y to guide the enumeration process. The set of symbols is empty. The grammar consists of the rules

$$\begin{aligned} g_0 &::= \lambda x.\ g_1 \\ g_1 &::= x \mid \mathsf{f}\ g_1 \mid \mathrm{ite}(g_2, g_1, g_1) \\ g_2 &::= \mathsf{true} \mid \mathsf{false} \mid g_1 = g_1 \mid \neg\, g_2 \mid g_2 \wedge g_2 \mid g_2 \vee g_2 \end{aligned}$$

In the first iteration, MBQI-Enum enumerates terms generated by the grammar and tries them in sequence. When the enumeration reaches the term $\lambda x.\ x$, MBQI-Enum generates the substitution $\sigma = \{y \mapsto \lambda x.\ x\}$. The strategy then determines whether the instantiation of the quantified formula, obtained by applying σ to the subformula $\exists z.\ y\ z \neq \mathsf{f}\ z$, refutes the current model. This is

achieved by checking whether the negation of that subformula, under the current substitution σ, is $\mathcal{T}$-satisfiable. Now, if we negate the subformula, instantiate y with $\lambda x.\ x$ in it and β-reduce, we obtain a formula equivalent to $\forall z.\ z = \mathsf{f}\ z$, which is satisfiable in $\mathcal{T}$ since in that theory the symbol f can be interpreted as the identity function. The substitution σ is then returned.

Back in the SMT loop, the instantiation lemma $(\forall y.\ \exists z.\ y\ z \neq \mathsf{f}\ z) \Longrightarrow \exists z.\ z \neq \mathsf{f}\ z$ is then added to F. In essence, this causes the solver to also add the propositionally entailed formula $\exists z.\ z \neq \mathsf{f}\ z$ to F. Since that formula has an existential quantifier prefix, it is Skolemized into $\mathsf{sk}_1 \neq \mathsf{f}\ \mathsf{sk}_1$ where sk_1 is a fresh symbol of type u. Then, the Skolemization lemma $(\exists z.\ z \neq \mathsf{f}\ z) \Longrightarrow \mathsf{sk}_1 \neq \mathsf{f}\ \mathsf{sk}_1$ added to F.

In the next iteration, MBQI-Enum tries the term $\lambda x.\ \mathsf{f}\ x$ generated by the grammar and produces the substitution $\sigma = \{y \mapsto \lambda x.\ \mathsf{f}\ x\}$, leading to the addition of the instantiation lemma $(\forall y.\ \exists z.\ y\ z \neq \mathsf{f}\ z) \Longrightarrow \exists z.\ \mathsf{f}\ z \neq \mathsf{f}\ z$ to F. Then, the formula $\exists z.\ \mathsf{f}\ z \neq \mathsf{f}\ z$ is Skolemized, with variable z replaced by a fresh Skolem symbol sk_2, and the Skolemization lemma $(\exists z.\ \mathsf{f}\ z \neq \mathsf{f}\ z) \Longrightarrow \mathsf{f}\ \mathsf{sk}_2 \neq \mathsf{f}\ \mathsf{sk}_2$ is added to F. Now, from the formulas $\forall y.\ \exists z.\ y\ z \neq \mathsf{f}\ z$, $(\forall y.\ \exists z.\ y\ z \neq \mathsf{f}\ z) \Longrightarrow \exists z.\ \mathsf{f}\ z \neq \mathsf{f}\ z$, and $(\exists z.\ \mathsf{f}\ z \neq \mathsf{f}\ z) \Longrightarrow \mathsf{f}\ \mathsf{sk}_2 \neq \mathsf{f}\ \mathsf{sk}_2$ in F, the quantifier-free subsolver deduces the propositionally entailed formula $\mathsf{f}\ \mathsf{sk}_2 \neq \mathsf{f}\ \mathsf{sk}_2$, which it then determines to be unsatisfiable. The SMT solver then concludes that the original F is unsatisfiable in $\mathcal{T}$, and hence that the conjecture is valid. ■

3 The Extended Strategy

Our strategy is based on MBQI-Enum, which has proved effective for higher-order reasoning. It adds three main ingredients:

- the grammar guiding the enumeration is augmented to include ε terms;
- fresh Skolem symbols are introduced to represent these terms; and
- term filtering is applied to eliminate redundant or invalid candidate terms during enumeration.

In addition to these extensions, our strategy generalizes MBQI-Enum by returning not only a substitution but also a set of lemmas. These lemmas are added to the input problem F, extending it iteratively.

The generalized strategy is detailed in Algorithm 3. It takes as input a quantified formula q of the form $\forall y_1, \ldots, y_m.\ \psi$ occurring in F, where ψ does not start with a universal quantifier. It starts by computing an initial substitution σ from the model (line 3). If no such substitution exists, it returns the empty set, indicating that no instantiation could be found.

After computing σ, the strategy iterates over the quantified variables $y_1, \ldots, y_m$ in the formula q to instantiate. For each y_i where $i \in \{1, \ldots, m\}$, it constructs a SyGuS grammar G_i based on uninterpreted symbols appearing in the entire input formula (line 7). The grammar is further augmented to include terms of the form $\varepsilon x.\ \varphi$, where x is a variable whose type is the return type of y_i and φ is a formula over x and other free variables and symbols from the formula q.

Algorithm 3 Generalized MBQI-Enum

```
1: function MBQI_ENUM_WITH_CHOICE(q)
2:     assume q is ∀y_1, ..., y_n. ψ
3:     let σ ← INSTS_MBQI(q)
4:     if σ does not exist then
5:         return ∅
6:     for each i ∈ {1, ..., n} do
7:         let G_i ← MAKE_GRAMMAR(q, y_i)
8:         for each j ∈ {1, 2, ...} do
9:             let e ← GET_ENUM_TERM(G_i, j)
10:            if e does not exist then
11:                break
12:            σ' ← σ[y_i ↦ e]
13:            if ¬ψσ' is satisfiable then
14:                σ ← σ'
15:                break
16:    return GET_LEMMAS(σ)
```

This grammar guides the enumeration of candidate terms that may be used to instantiate y_i. As a result, the SyGuS enumerator generates both ε and non-ε terms as potential substitutions for y_i.

For each candidate term e generated by the enumerator, a new substitution σ' is formed by mapping y_i to the current term e in the initial substitution σ (line 12). The strategy then checks whether the negation of the body of the quantified formula under σ' is satisfiable in the current model (line 13). This satisfiability check serves as a semantic filter, pruning out terms that do not yield counterexamples. If the check succeeds, σ is updated and the enumeration process proceeds to the next quantified variable. Otherwise, it continues with the next candidate term. Once all quantified variables have been considered, our strategy returns both the final substitution σ and a set of lemmas, which include those introduced during the handling of ε terms and quantifier instantiations. These lemmas are subsequently added to the input formula F, extending it iteratively.

Augmented Grammar. Our strategy augments the base grammar used for term enumeration in MBQI-Enum by incorporating ε terms, allowing the generation of arbitrary formulas. Our approach is detailed in Algorithm 4.

MBQI-Enum constructs a base grammar for each universally quantified variable y_i in the formula q using a set of symbols S from the entire input formula, such as uninterpreted symbols and bound variables (line 4). A grammar G is a triple (g_0, G, R_G), where g_0 is the initial symbol, G is a set of nonterminal symbols with $g_0 \in G$, and R_G is a set of production rules. Each rule has the form $g ::= t$, where $g \in G$ and t is a term built from symbols in the set S, nonterminals in G, free variables, and symbols from the signature of a background theory $\mathcal{T}$. The symbols in S, along with the free variables and the theory symbols from $\mathcal{T}$, act as terminal symbols in the grammar. Each grammar generates

Algorithm 4 Augmented grammar

```
 1: function MAKE_GRAMMAR(q, y_i)
 2:     let F be the input formula
 3:     S ← symbols(F) ∪ symbols(q) ∪ {y_{i+1}, ..., y_n}
 4:     let (g_0, G, R_G) ← MAKE_BASE_GRAMMAR(S, y_i)
 5:     if choice is enabled then
 6:         let x be a variable of type τ
 7:         (p_0, P, R_P) ← MAKE_PREDICATE_GRAMMAR(S, x, y_i)
 8:         extend G ← G ∪ P
 9:         extend R_G ← {g_0 ← εx. p_0} ∪ R_G ∪ R_P
10:     return (g_0, G, R_G)
11: function MAKE_PREDICATE_GRAMMAR(S, x, y_i)
12:     S ← S ∪ {x}
13:     let z̄ ← bound variables in y_i
14:     define p_0 nonterminals
15:     let P ← {z̄} ∪ S ∪ {true, false, =, ¬, ∧}
16:     define R_P ← production rules
17:     return (p_0, P, R_P)
```

terms whose type matches the return type τ of y_i. For higher-order variables, the grammars include rules for generating λ-abstractions. In this case, the initial symbol g_0 expands to λ-terms. Whereas earlier versions of MBQI-Enum considered only terms in η-long β-normal form [19], the current cvc5 implementation allows partial application. The variables bound by λ-abstractions are terminal symbols in the grammar that generates the abstraction's body. For example, if y_i is a function variable of arity n whose arguments and result are all of the same type, the grammar includes rules such as

$$g_0 ::= \lambda x_1, \ldots, x_n.\ g_1 \qquad\qquad g_1 ::= x_1 \mid \cdots \mid x_n$$

To include ε terms of the form $\varepsilon x.\ \varphi$, our strategy extends the grammar as follows. For the nonterminal g_1 in the base grammar corresponding to y_i's return type τ, a variable x of type τ is introduced (line 6) and added to the symbol set S (line 12). A new grammar is then built to generate Boolean formulas over the extended symbol set $S \cup \{x\}$ (line 7). This predicate grammar is defined as the triple (p_0, P, R_P), where p_0 is the initial nonterminal symbols and P is a set of nonterminal symbols that includes p_0. The set R_P contains production rules of the form $p ::= t$, where $p \in P$ and t is a term. Each term t is built from symbols in S, nonterminals in P, variables that were bound in y_i but now appear free, and symbols from the signature of the Boolean background theory (line 15). These elements act as terminal symbols in the grammar. The new grammar includes the rules

$$p_0 ::= \mathsf{true} \mid \mathsf{false} \mid p_1 = p_1 \mid \neg p_0 \mid p_0 \wedge p_0 \qquad p_1 ::= x_1 \mid \cdots \mid x_n \mid x$$

The nonterminal p_0 directly defines the body of the ε term. We define a new production rule of the form $g_1 ::= \varepsilon x.\ p_0$. The resulting grammar is con-

structed by extending the base grammar (g_0, G, R_G) with the predicate grammar (p_0, P, R_P) and the new production rule (lines 8–9). The final grammar is defined as $(p_0, G \cup P, \{g_1 ::= \varepsilon x.\ p_0\} \cup R_G \cup R_P)$. It can be used to enumerate both ε and non-ε terms and is thus more expressive than the original grammar. For our example, the final grammar consists of the rules

$$g_0 ::= \lambda x_1, \ldots, x_n.\ g_1 \qquad g_1 ::= \varepsilon x.\ p_0 \mid x_1 \mid \cdots \mid x_n$$
$$p_0 ::= \mathsf{true} \mid \mathsf{false} \mid p_1 = p_1 \mid \neg p_0 \mid p_0 \wedge p_0 \qquad p_1 ::= x_1 \mid \cdots \mid x_n \mid x$$

Skolem Symbols. The most important part of the strategy is where we take formulas containing ε terms generated by the grammar, abstract these terms into fresh Skolem symbols, and generate corresponding lemmas with triggers to guide efficient higher-order quantifier instantiation. Our approach is detailed in Algorithm 5.

Algorithm 5 Lemmas for Skolem symbols

```
 1: function GET_LEMMAS(σ)
 2:     let L ← ∅
 3:     let ψ′ ← ψ
 4:     for all i ∈ {1, ..., m} do
 5:         assume y_i σ = λ(x_1, ..., x_n). t        ▷ y_i is the variable being instantiated
 6:         let t′ ← t
 7:         for all subterms e of t of the form εx : τ. φ(z̄, x) do
 8:             let z̄ = (z_1 : τ_1, ..., z_k : τ_k) ⊆ {x_1, ..., x_n} be φ's free variables
 9:             let h : (τ_1, ..., τ_k) → τ be a fresh Skolem symbol
10:             let lemma ← ∀z̄. ¬ (∃x. φ(z̄, x)) ∨ φ(z̄, h z̄)
11:             extend L ← L ∪ {lemma}
12:             replace e in t′ with h z̄
13:         ψ′ ← ψ′{y_i ↦ λ(x_1, ..., x_n). t′}
14:     let inst_lemma ← (∀y_1, ..., y_m. ψ) ⟹ ψ′
15:     extend L ← L ∪ {inst_lemma}
16:     return L
```

Our strategy generalizes MBQI-Enum by iterating over the variables in $\forall y_1, \ldots, y_m.\ \psi$ and incrementally building a fully instantiated formula before generating the instantiation lemmas. It then returns the set of generated lemmas L instead of only the substitutions.

Given a substitution $\sigma = \{y_i \mapsto \lambda x_1, \ldots, x_n.\ t\}$ for each variable y_i, consider any subterm e of t of the form $\varepsilon x : \tau.\ \varphi(\bar{z}, x)$, where $\bar{z} = (z_1 : \tau_1, \ldots, z_k : \tau_k)$, the free variables of e, are included in $\{x_1, \ldots, x_n\}$. We replace all occurrences of e in t with an application of a fresh Skolem symbol h to the variables $\bar{z}$ (lines 5–10). In other words, the subterm $\varepsilon x.\ \varphi(\bar{z}, x)$ is converted to $\mathsf{h}\ \bar{z}$, where h has type $\tau_1 \to \cdots \to \tau_k \to \tau$. The introduction of these Skolems abstracts the ε binder into symbols understood by the rest of the SMT solver. Our strategy constructs the lemma $\forall \bar{z}.\ \neg\,(\exists x.\ \varphi(\bar{z}, x)) \vee \varphi(\bar{z}, \mathsf{h}\ \bar{z})$ (line 10) and adds it to the

lemma set L. When later added to the set F, this lemma makes the strategy less incomplete while preserving F's satisfiability. It guarantees that if there exists an x satisfying $\varphi(\bar{z}, x)$, then $\mathsf{h}\ \bar{z}$ satisfies φ as well.

Additionally, we mark the Skolem application $\mathsf{h}\ \bar{z}$ as a *trigger* [12,13] in the lemma. This trigger guides E-matching, ensuring that instantiations are driven by terms that match the structure of the Skolem application, which is applied to exactly the same set of variables as in the original quantified formula.

After all ε-subterms in t have been abstracted, the modified substitution $y_i \mapsto \lambda x_1, \ldots, x_n.\ t'$, where t' is obtained by replacing the ε-subterms of t with the corresponding Skolem applications $\mathsf{h}\ \bar{z}$ (line 12), is applied to the partially instantiated body ψ of the formula q (line 13). This procedure is repeated for each variable y_i, yielding a fully instantiated formula ψ' in which all substitutions have been applied to the universally quantified variables and all Skolem symbols have been introduced. Finally, the instantiation lemma $(\forall y_1, \ldots, y_m.\ \psi) \Longrightarrow \psi'$ is generated (line 14) and added to the lemma set L. The complete set L is eventually returned and merged with the current set F.

Filtering. To avoid redundant instantiations, term filtering is applied during enumeration. Enumerated terms already present in the set of candidate terms are discarded, ensuring that terms equivalent under rewriting are considered only once. For terms of the form $\varepsilon x.\ \varphi$, the bound variable x must occur in the body φ. Terms in which x does not occur are excluded, to prevent the introduction of trivial instantiations. Since cvc5 already performs extensive theory-specific filtering by default, no additional filtering is required in our setting.

Example 6. We return to Example 1 and present it in more detail. Let $\mathcal{T}$ be the same theory as in the previous examples. Let u be an uninterpreted sort and let f be an uninterpreted function of type $u \to u$. Consider the conjecture

$$(\forall x, y.\ \mathsf{f}\ x = \mathsf{f}\ y \Longrightarrow x = y) \Longrightarrow \exists g.\ \forall z.\ g\ (\mathsf{f}\ z) = z$$

where g has type $u \to u$ and x, y, z have type u. The formula states that if f is injective, then it has a left inverse. We prove that the formula holds in $\mathcal{T}$ by contradiction. That is, we negate it, obtaining the following input problem

$$F = \{\forall x, y.\ \mathsf{f}\ x = \mathsf{f}\ y \Longrightarrow x = y,\ \forall g.\ \exists z.\ g\ (\mathsf{f}\ z) \neq z\}\ ,$$

and prove that F is unsatisfiable in $\mathcal{T}$.

MBQI-Enum generates a grammar for g using the set of symbols $S = \{\mathsf{f}\}$. The grammar consists of the rules

$$g_0 ::= \lambda y.\ g_1 \qquad\qquad g_1 ::= y \mid \mathsf{f}\ g_1$$

Based on this grammar, MBQI-Enum generates the substitutions $\{g \mapsto \lambda y.\ y\}$, $\{g \mapsto \lambda y.\ \mathsf{f}\ y\}$, $\{g \mapsto \lambda y.\ \mathsf{f}\ (\mathsf{f}\ y)\}$, ... for g. Since this enumeration does not capture inverse functions, the SMT solver does not terminate.

Our strategy addresses this challenge by constructing a richer grammar. We introduce a variable x of type u and extend the symbol set to $S \cup \{x\}$. We then define the following grammar:

$$p_0 ::= \mathsf{true} \mid \mathsf{false} \mid p_1 = p_1 \mid \neg p_0 \mid p_0 \wedge p_0 \qquad p_1 ::= y \mid x \mid \mathsf{f}\ p_1$$

We extend the MBQI-Enum grammar by adding a rule to produce $\varepsilon x.\ p_0$. The result consists of the rules for p_0 and p_1 above and the following rules:

$$g_0 ::= \lambda y.\ g_1 \qquad g_1 ::= \varepsilon x.\ p_0 \mid y \mid \mathsf{f}\ g_1$$

Using this grammar, our strategy generates the term $\lambda y.\ \varepsilon x.\ y = \mathsf{f}\ x$, which denotes a left inverse of f, leading to the substitution $\{g \mapsto \lambda y.\ \varepsilon x.\ y = \mathsf{f}\ x\}$.

Corresponding to this term, we introduce a fresh Skolem symbol $\mathsf{h} : u \to u$. Our strategy asserts the following lemma:

$$\forall y.\ \neg\,(\exists x.\ y = \mathsf{f}\ x) \vee y = \mathsf{f}\ (\mathsf{h}\ y) \tag{ℓ_1}$$

The lemma states that for any y, if there exists an x such that $y = \mathsf{f}\ x$ (i.e., y is in the image of f), then applying f to $\mathsf{h}\ y$ must yield y. The original enumerated term $\lambda y.\ \varepsilon x.\ y = \mathsf{f}\ x$ is thus rewritten to the term $\lambda y.\ \mathsf{h}\ y$. The substitution $\sigma = \{g \mapsto \lambda y.\ \mathsf{h}\ y\}$ is considered. Next, the instantiation lemma

$$(\forall g.\ \exists z.\ g\ (\mathsf{f}\ z) \neq z) \Longrightarrow \exists z.\ \mathsf{h}\ (\mathsf{f}\ z) \neq z \tag{ℓ_2}$$

is generated by applying the substitution σ followed by β-reduction. The set of lemmas $L = \{(\ell_1), (\ell_2)\}$ is constructed. Each lemma in L is then added to set F. After that, Skolemization introduces a fresh symbol sk corresponding to the existentially quantified variable z. The lemma

$$(\exists z.\ \mathsf{h}\ (\mathsf{f}\ z) \neq z) \Longrightarrow \mathsf{h}\ (\mathsf{f}\ \mathsf{sk}) \neq \mathsf{sk} \tag{ℓ_3}$$

is also added to F.

The strategy proceeds to instantiate the universally quantified formulas in F. Quantified variables of type u are instantiated using terms from F. First, the strategy instantiates x and y in $\forall x, y.\ \mathsf{f}\ x = \mathsf{f}\ y \Longrightarrow x = y$ with sk and $\mathsf{h}\ (\mathsf{f}\ \mathsf{sk})$, respectively. Then it instantiates x and y in the lemma $\forall y.\ \neg\,(\exists x.\ y = \mathsf{f}\ x) \vee y = \mathsf{f}\ (\mathsf{h}\ y)$ with sk and $\mathsf{f}\ \mathsf{sk}$, respectively. These instantiations arise from E-matching, which matches each lemma's pattern against the terms already present in the equivalence classes maintained by the SMT solver. The instantiation lemmas

$$(\forall y.\ \neg\,(\exists x.\ y = \mathsf{f}\ x) \vee y = \mathsf{f}\ (\mathsf{h}\ y)) \Longrightarrow \mathsf{f}\ \mathsf{sk} \neq \mathsf{f}\ \mathsf{sk} \vee \mathsf{f}\ \mathsf{sk} = \mathsf{f}\ (\mathsf{h}\ (\mathsf{f}\ \mathsf{sk})) \tag{ℓ_4}$$

$$(\forall x, y.\ \mathsf{f}\ x = \mathsf{f}\ y \Longrightarrow x = y) \Longrightarrow \mathsf{f}\ \mathsf{sk} \neq \mathsf{f}\ (\mathsf{h}\ (\mathsf{f}\ \mathsf{sk})) \vee \mathsf{sk} = \mathsf{h}\ (\mathsf{f}\ \mathsf{sk}) \tag{ℓ_5}$$

are then added to F. At this point, the quantifier-free subsolver finds F unsatisfiable in $\mathcal{T}$ by determining that the conclusions of the lemmas ℓ_3, ℓ_4, and ℓ_5 are jointly unsatisfiable in $\mathcal{T}$. This implies that the conjecture is valid. ■

Example 7. We consider a simplified version of TPTP benchmark `SYO268^5`. Let u_1, u_2 be uninterpreted sorts, and let r be a function symbol of type $u_1 \to u_2 \to o$. Consider the conjecture

$$(\forall x.\ \exists y.\ \mathsf{r}\ x\ y) \Longrightarrow \exists g.\ \forall x.\ \mathsf{r}\ x\ (g\ x)$$

where g has type $u_1 \to u_2$, x has type u_1, and y has type u_2. The formula states that for every total relation r on $u_1 \times u_2$, there exists a corresponding function from u_1 to u_2. To prove the formula by contradiction, we negate it and obtain the input problem $F = \{\forall x.\ \exists y.\ \mathsf{r}\ x\ y,\ \forall g.\ \exists x.\ \neg\, \mathsf{r}\ x\ (g\ x)\}$.

MBQI-Enum generates a grammar for g using the set of symbols $S = \{\mathsf{r}\}$. The grammar consists of the rules

$$g_0 ::= \lambda y.\ g_1 \qquad\qquad g_1 ::= t$$

Based on this grammar, MBQI-Enum generates the substitution $\{g \mapsto \lambda y.\ t\}$, where t is a ground term of type u_2. As a result, the SMT solver terminates with an unknown status.

By contrast, with our extension, the solver behaves as follows. Let x be a variable of type u_2. We extend the symbol set to $S \cup \{x\}$. Consider the rules

$$\begin{aligned} p_0 &::= \mathsf{true} \mid \mathsf{false} \mid p_1 = p_1 \mid p_2 = p_2 \mid \mathsf{r}\ p_1\ p_2 \mid \neg p_2 \mid p_2 \wedge p_2 \\ p_1 &::= y \\ p_2 &::= x \end{aligned}$$

We augment the MBQI-Enum grammar by adding a rule to produce $\varepsilon x.\ p_0$. The resulting grammar consists of the rules for p_0, p_1, and p_2 above together with

$$g_0 ::= \lambda y.\ g_1 \qquad\qquad g_1 ::= \varepsilon x.\ p_0 \mid t$$

Using this grammar, our strategy generates the term $\lambda y.\ \varepsilon x.\ \mathsf{r}\ y\ x$, leading to the substitution $\{g \mapsto \lambda y.\ \varepsilon x.\ \mathsf{r}\ y\ x\}$.

Corresponding to this term, we introduce a fresh Skolem symbol $\mathsf{h} : u_1 \to u_2$. Our strategy asserts the following lemma:

$$\forall y.\ \neg\,(\exists x.\ \mathsf{r}\ y\ x) \vee \mathsf{r}\ y\ (\mathsf{h}\ y) \tag{ℓ_1}$$

The original enumerated term $\lambda y.\ \varepsilon x.\ \mathsf{r}\ y\ x$ is thus rewritten to the term $\lambda y.\ \mathsf{h}\ y$. The substitution $\sigma = \{y \mapsto \lambda y.\ \mathsf{h}\ y\}$ is considered. Next, the instantiation lemma

$$(\forall g.\ \exists x.\ \neg\, \mathsf{r}\ x\ (g\ x)) \Longrightarrow \exists x.\ \neg\, \mathsf{r}\ x\ (\mathsf{h}\ x) \tag{ℓ_2}$$

is generated by applying the substitution σ followed by β-reduction. The set of lemmas $L = \{(\ell_1), (\ell_2)\}$ is constructed and then merged with F. Afterward, Skolemization introduces a fresh symbol sk corresponding to the existentially quantified variable x. The lemma

$$(\exists x.\ \neg\, \mathsf{r}\ x\ (\mathsf{h}\ x)) \Longrightarrow \neg\, \mathsf{r}\ \mathsf{sk}\ (\mathsf{h}\ \mathsf{sk}) \tag{ℓ_3}$$

is added to F.

The strategy proceeds to instantiate the universally quantified formulas of F. Quantified variables of nonfunction type are instantiated using terms from F. First, the strategy instantiates x and y in the lemma $\forall y.\ \neg\,(\exists x.\ \mathsf{r}\ y\ x) \vee \mathsf{r}\ y\ (\mathsf{h}\ y)$ with $\mathsf{h}\ \mathsf{sk}$ and sk, respectively. Then it instantiates x and y in the axiom $\forall x.\ \exists y.\ \mathsf{r}\ x\ y$ with sk, and $\mathsf{h}\ \mathsf{sk}$, respectively. These instantiations arise from E-matching, which matches each lemma's pattern against the terms already present in the equivalence classes maintained by the solver. The instantiation lemmas

$$(\forall y.\ \neg\,(\exists x.\ \mathsf{r}\ y\ x) \vee \mathsf{r}\ y\ (\mathsf{h}\ y)) \Longrightarrow \neg\,\mathsf{r}\ \mathsf{sk}\ (\mathsf{h}\ \mathsf{sk}) \vee \mathsf{r}\ \mathsf{sk}\ (\mathsf{h}\ \mathsf{sk}) \qquad (\ell_4)$$

$$(\forall x.\ \exists y.\ \mathsf{r}\ x\ y) \Longrightarrow \mathsf{r}\ \mathsf{sk}\ (\mathsf{h}\ \mathsf{sk}) \qquad (\ell_5)$$

are added to F. At this point, the quantifier-free subsolver finds F unsatisfiable. Hence the conjecture is valid. ■

4 Implementation

We developed our strategy as an extension of cvc5's implementation of MBQI-Enum. Our extension refines the selection of grammars for term enumeration, introduces a mechanism for managing choice terms generated by these grammars, and incorporates a filtering step to discard redundant or unsuitable terms.

For each quantified variable, MBQI-Enum constructs a SyGuS grammar that specifies the space of candidate instantiations (Algorithm 4, line 4). Our implementation augments these grammars with terminal rules for generating choice terms. Specifically, for each nonterminal type that qualifies, we add a corresponding expression of the form $\varepsilon x.\ \varphi$, which selects a value satisfying a predicate over that type (lines 5–9). This allows the enumerator to synthesize Skolem-style instantiations directly from the grammar.

We use the existing implementation of the fast SyGuS-based enumerator for term generation. This enumerator considers only terms that are unique up to rewriting [31]. During the enumeration process, ε terms are constructed within cvc5's internal abstract syntax tree representation, thereby exploiting cvc5's built-in rewriter. For example, $\varepsilon x.\ x = t$, where x does not occur free in t, is rewritten to t.

Beyond these rewrite-based simplifications, our implementation applies additional heuristics for filtering ε terms. In particular, it explicitly discards any term of the form $\varepsilon x.\ \varphi$, where x does not occur free in φ. These heuristics are implemented via a callback from the SyGuS enumerator.

The enumerator constructs ε terms that are immediately purified by our implementation. Recall that each ε term is replaced with a fresh Skolem symbol, and an accompanying lemma is generated to relate the Skolem to the original ε term (Algorithm 5, lines 6–10). For example, given a term $\varepsilon x.\ \varphi$, the procedure introduces a fresh Skolem h and asserts that either $\varphi(\mathsf{h})$ holds or the quantified condition that motivated the ε term is already satisfied. An important heuristic is that our approach designates the Skolem application itself as a trigger for the

generated quantified formula, thereby ensuring that instantiations during solving are guided by the intended terms.

During instantiation, terms containing ε terms are first normalized through the purification and filtering procedures described above before being considered as candidate substitutions. The generalized strategy, shown in Algorithm 3, iteratively builds substitutions by enumerating terms from the augmented grammars (lines 6–12), tests them in the current model (line 13), and commits suitable assignments (line 14). Notably, unlike the standard MBQI-Enum procedure, our strategy returns not only substitutions but also auxiliary lemmas (line 16).

5 Evaluation

We extensively evaluated our cvc5 implementation of the extended MBQI-Enum on higher-order benchmarks.

Since the publication of Kondylidou et al. [19], MBQI-Enum has been developed further independently of our work on choice. In our evaluation, by "original MBQI-Enum" we mean the most recent version of the strategy excluding our modifications related to ε. This strategy includes the following enhancements over Kondylidou et al.:

- MBQI-Enum now supports partial applications, allowing the strategy to consider candidate terms that partially apply functions during enumeration.
- Timeouts have been added to subsolver invocations used for checking candidate instantiations, preventing individual candidates from stalling the overall SMT loop. In addition, there is an option that completely prohibits nested MBQI calls, which avoids potential nontermination caused by nested subsolver invocations.
- MBQI-Enum now incorporates a guess-and-test loop with a progress-aware instantiation cache: When a proposed instantiation fails to refute the current model, it is cached to avoid rechecking the same substitution later, thereby ensuring that the strategy always makes progress.
- We now incorporate a lookahead approach that preemptively considers Skolem symbols for quantified formulas that are not introduced during preprocessing. These symbols are incorporated into the SyGuS grammars, thus enriching the set of terms we consider for instantiation with MBQI-Enum.

Setup. We denote our configuration of MBQI-Enum extended with choice by cvc5[C]. We compare our strategy against several established quantifier instantiation strategies in cvc5: cvc5[s], which uses SyQI [27]; cvc5[m], which implements MBQI [15]; cvc5[M], which implements the original MBQI-Enum [19]; cvc5[hoelim], which eagerly rewrites higher-order constraints into first-order form and applies full quantifier saturation; and cvc5[foinst], which relies on exhaustive first-order-style instantiation and axiomatic handling of higher-order applications.

Table 1. Extended MBQI-Enum vs. other provers on TPTP TH0 benchmarks

	Satallax	Vampire	Zipperposition	cvc5[C]
Satisfiable	196	14	0	**204**
Unsatisfiable	2162	**2303**	2083	2178
Total	2358	2317	2083	**2382**
Unknown	15	16	0	154
Timeouts	1384	1424	1674	1221

We also include a comparison with the state-of-the-art provers Satallax [10], Vampire [7], and Zipperposition [40]. Satallax was run with its default settings. Vampire was run in portfolio mode, as its higher-order reasoning configuration automatically enables this mode and disabling it makes the prover much less competitive. For Zipperposition, we enabled several options that were extensively evaluated by Bozec and Blanchette [9] and shown to achieve the best overall performance on higher-order benchmarks.

All experiments were conducted on a machine with a 40-core Intel Xeon Silver 4114 CPU running at 2.20 GHz, equipped with 192 GB of RAM and running Debian 12 (Bookworm). Each benchmark was executed with a timeout of 60 seconds.

Benchmarks. The experiments were carried out on monomorphic higher-order problems (TH0) from version 9.1.0 of the TPTP library [39]. The benchmark set consists of 3757 problems. From the 3962 TH0 problems, we excluded 205 benchmarks that one or more systems could not parse (e.g., because of the use of interpreted arithmetic).

To support reasoning with choice, we implemented external parser support for the TPTP operators for ε and the definite description operator ι. Although the TPTP library contains very few problems with ε and ι, they might arise in users' problems.

Results. The results are shown in Tables 1 and 2, where bold indicates the most successful system. Notably, our approach achieves the highest total count of solved benchmarks, surpassing the nearest competitor by 24 solved problems and the baseline MBQI-Enum by 23.

Table 1 compares our strategy (cvc5[C]) with external state-of-the-art provers. Our approach achieves the highest total number of solved benchmarks, with 2382 problems, surpassing Satallax, Vampire, and Zipperposition. It outperforms the nearest competitor by 24 benchmarks. In particular, our strategy solves 204 satisfiable problems, while Satallax solves 196, Vampire solves 14, and Zipperposition solves none. For unsatisfiable problems, Vampire performs the best, solving 2303 benchmarks, whereas our strategy solves 2178, Satallax 2162, and Zipperposition 2083. Overall, our strategy demonstrates strong performance across

Table 2. Extended MBQI-Enum vs. other cvc5 strategies on TPTP TH0 benchmarks

	cvc5[hoelim]	cvc5[foinst]	cvc5[s]	cvc5[m]	cvc5[M]	cvc5[C]
Satisfiable	14	37	88	188	204	**204**
Unsatisfiable	2082	2126	1657	2074	2155	**2178**
Total	2096	2163	1745	2262	2359	**2382**
Unknown	0	125	45	289	158	154
Timeouts	1661	1469	1967	1206	1240	1221

both satisfiable and unsatisfiable problems compared with the state-of-the-art provers.

Table 2 compares cvc5[C] with other cvc5 quantifier instantiation strategies. Our configuration solves 23 unsatisfiable benchmarks that the baseline MBQI-Enum cannot, with no benchmarks lost, while solving the same number of satisfiable problems (204). This shows that extending MBQI-Enum with choice consistently improves solver performance. Overall, when compared against all other available quantifier instantiation strategies, our strategy achieves the strongest performance, demonstrating that it is considerably more effective on higher-order benchmarks than any other individual technique in cvc5. It may seem as though 23 additional benchmarks are not that many; however, we should bear in mind that the vast majority of higher-order TPTP benchmarks do not contain choice and can likely be solved without reasoning about choice.

In addition, our technique solves several problems not solved by its competitors: 143 problems not solved by Vampire, 296 not solved by Zipperposition, 235 not solved by Satallax, and 21 not solved by any other cvc5 strategy. When considering all systems, cvc5[C] solves exactly one benchmark that no other system can solve.

6 Related Work

Besides MBQI-Enum, other quantifier instantiation strategies work on higher-order problems, notably E-matching, MBQI, and SyQI. In higher-order E-matching, quantified variables are instantiated by matching their patterns against ground terms available in the current context—that is, the set of ground terms introduced during proof search. This technique has been extended to support higher-order features, such as functional arguments [3]. Higher-order MBQI constructs candidate models that provide interpretations for functions. This enables the solver to generate ground instantiations of quantified formulas in an attempt to refute the current model [15]. Higher-order SyQI further improves the instantiation process by selecting candidate terms using syntactic templates and heuristics derived from the problem's structure, guided by a fixed grammar [27]. None of these approaches provides special support for Hilbert's choice.

Other higher-order provers support Hilbert's choice in various ways. Satallax implements specialized tableau inference rules that directly reason about ε

terms in a sound and refutationally complete way [2]. Leo-III integrates ε into its extensional higher-order paramodulation calculus and supports Henkin semantics with choice [38]. Zipperposition includes a dedicated choice axiom within its higher-order superposition calculus, allowing it to reason about ε terms as needed [6]. Vampire supports Hilbert's choice either through a Leo-III-style inference or by introducing the choice axiom [7].

Hilbert's choice is also present in proof assistants such as Isabelle/HOL [28], Lean [24], and Rocq [36] (formerly known as Coq). In Isabelle/HOL, the choice operator is introduced as an axiom as part of the system's core libraries. In systems based on dependent type theory, such as Lean and Rocq, choice is introduced via classical axioms, which should be avoided in constructive definitions and proofs. Even in Isabelle/HOL, some users, such as Paulson [29], prefer to avoid choice:

> Pragmatists may argue that in verification nobody cares whether choice is used or not. However, pragmatists should be concerned that reasoning about ε-terms is tricky.

Reasoning about choice is indeed tricky, but cvc5 can do it reasonably well now.

7 Conclusion

We presented an SMT quantifier instantiation strategy that extends the MBQI-Enum strategy so as to reason natively about Hilbert's choice operator. It can be used to solve problems that contain Hilbert's choice but also problems that do not (cf. Example 1). We implemented the approach in the SMT solver cvc5. The empirical results on higher-order TPTP benchmarks show that our strategy is clearly superior to the original MBQI-Enum.

We believe there is potential for further improving the strategy. We noticed that the ε terms we need often occur too late in the term enumeration (which is by increasing term size), since they tend to be relatively complex. Better term enumeration heuristics and term filtering could help the SMT solver generate the desired terms earlier—before the timeout.

Acknowledgments. We thank Mark Summerfield and the anonymous reviewers for their helpful comments. We also thank Haniel Barbosa, who provided many comments on an earlier draft.

This research was cofunded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

This research was also partially funded by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

Data-Availability Statement. The artifact associated with this paper, including the software and scripts used for experiments, is publicly available on Zenodo [20]. The raw evaluation data are also available on Zenodo [21]. The instructions on the first Zenodo page [20] explain how to reproduce the experimental evaluation.

References

1. Andrews, B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, APLS, vol. 27. Springer (2022)
2. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. Journal of Automated Reasoning **47**, 451–479 (2011)
3. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fishman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer (2022)
4. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 2019. LNCS, vol. 11716, pp. 35–54. Springer (2019)
5. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer (2011)
6. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for higher-order logic. Journal of Automated Reasoning **67** (2023)
7. Bhayat, A., Suda, M.: A higher-order Vampire (short paper). In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) IJCAR 2024. LNCS, vol. 14739, pp. 75–85. Springer (2024)
8. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 151–156. Springer (2009)
9. Bozec, T., Blanchette, J.: Iterative monomorphisation. In: Thiemann, R., Weidenbach, C. (eds.) FroCoS 2025. LNCS, vol. 15979, pp. 269–286. Springer (2025)
10. Brown, C.E.: Satallax: An automatic higher-order prover. Journal of Automated Reasoning **7364**, 111–117 (2012)
11. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**, 56–68 (1940)
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52**(3), 365–473 (2005)
13. Dross, C., Conchon, S., Paskevic, A.: Reasoning with triggers. In: Fontaine, P., Goel, A. (eds.) SMT 2012. EPiC Series in Computing, vol. 20, pp. 22–31 (2012)
14. Ge, Y., Barrett, C.W., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. Annals of Mathematics and Artificial Intelligence **55**(1-2), 101–122 (2009)
15. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer (2009)
16. Gordon, M.J.C.: Introduction to the HOL system. In: Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J. (eds.) HOL 1991. pp. 2–3. IEEE (1991)

17. Hilbert, D., Ackermann, W.: Grundzüge der Theoretischen Logik, Grundlehren der mathematischen Wissenschaften, vol. 27. Springer (1928)
18. Hilbert, D., Bernays, P.: Grundlagen der Mathematik I, Grundlehren der mathematischen Wissenschaften, vol. 40. Springer (1934)
19. Kondylidou, L., Reynolds, A., Blanchette, J.: Augmenting model-based instantiation with fast enumeration. In: Gurfinkel, A., Heule, M. (eds.) TACAS 2025. LNCS, vol. 15696, pp. 85–103. Springer (2025)
20. Kondylidou, L., Reynolds, A., Blanchette, J., Tinelli, C.: Artifact for paper "Enumerating choice terms in model-based quantifier instantiation". In: TACAS 2026. Zenodo (2026), https://doi.org/10.5281/zenodo.18304537
21. Kondylidou, L., Reynolds, A., Blanchette, J., Tinelli, C.: Enumerating choice terms in model-based quantifier instantiation [data set]. In: TACAS 2026. Zenodo (2026), https://doi.org/10.5281/zenodo.18212605
22. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 183–198. Springer (2007)
23. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
24. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS, vol. 12699, pp. 625–635. Springer (2021)
25. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) CAV 2023. LNCS, vol. 13965, pp. 3–17. Springer (2023)
26. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014)
27. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Syntax-guided quantifier instantiation. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12652, pp. 145–163. Springer (2021)
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
29. Paulson, L.C.: Defining functions on equivalence classes. In: Guruswami, V., Glazener, T. (eds.) CoRR 2019. vol. 7, pp. 658–675. Association for Computing Machinery (2019)
30. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D. (ed.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer (2018)
31. Reynolds, A., Barbosa, H., NÖtzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part II. LNCS, vol. 11562, pp. 74–83. Springer (2019)
32. Reynolds, A., Blanchette, J.C., Cruanes, S., Tinelli, C.: Model finding for recursive functions in SMT. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 133–151. Springer (2016)
33. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer (2015)
34. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 640–655. Springer (2013)
35. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: FMCAD 2014. pp. 195–202. IEEE (2014)
36. Rocq Development Team: Rocq Prover. https://zenodo.org/records/15149629 (2025), Zenodo

37. Simpson, S.G.: The epsilon calculus and conservative extensions. Notre Dame Journal of Formal Logic **25**(3), 193–202 (1984)
38. Steen, A.: Extensional Paramodulation for Higher-Order Logic and its Effective Implementation Leo-III. Ph.D. thesis, Freie Universität Berlin (2020)
39. Sutcliffe, G.: The TPTP problem library and associated infrastructure—from CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**, 483–502 (2017)
40. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. Journal of Automated Reasoning **66**, 541–564 (2022)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Tao's Equational Proof Challenge Accepted

Lydia Kondylidou[1], Jasmin Blanchette[1], and Marijn J.H. Heule[2]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
{l.kondylidou,jasmin.blanchette}@lmu.de
[2] Carnegie Mellon University, Pittsburgh, United States
marijn@cmu.edu

Abstract. In the context of the Equational Theories Project, Terence Tao posed the challenge of finding alternatives to a complicated 62-step proof found by the Vampire superposition prover. We introduce a proof minimization tool called Krympa. Using a combination of brute force and heuristics, and exploiting both Vampire and the Twee equational prover, the tool reduces the 62-step proof to 20 steps, each corresponding to a rewrite. In an empirical evaluation, it also performs well on 1431 equational problems originating from the same project, reducing in particular a 151-step proof to only 10 steps.

Keywords: Theorem provers · Equational logic · Proof minimization.

1 Introduction

The Equational Theories Project [6], launched in September 2024 by Fields medalist Terence Tao, aims at exploring the relations between different equational theories of magmas. A *magma* is a basic algebraic structure consisting of a set equipped with a single binary operation $\diamond$ closed on that set. The project's first phase, concluded in April 2025, focused on equational laws for magmas that contain at most four applications of $\diamond$.

The project uses the Lean [24] proof assistant to express proofs and counterexamples but depends on automatic theorem provers and other external tools. The problems explored in the project's first phase all fall within first-order logic's unit equality fragment: They consist of a universally quantified equation as the sole axiom and a universally quantified equation as the proof goal, or conjecture.

For the problem $650 \implies 448$, where 650 denotes the axiom $\forall x, y, z.\ x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ and 448 denotes the conjecture $\forall x, y, z.\ x = x \diamond (y \diamond (z \diamond (x \diamond z)))$, the Vampire [4] superposition prover found a particularly complex proof, with 62 inference steps, excluding clausification and Skolemization. Given that the proof is unintelligible, Tao challenged the community to find "an alternate proof, by whatever means you wish—human, semi-automated, or automated" [30].

One idea was to run a specialized equational prover, Twee [26], instead of Vampire, but this results in a very long, 137-step proof. Another approach would be to use Lean's automation, such as the `aesop` [21], `canonical` [25], `duper` [8], and `grind` [23] tactics and LeanHammer [32], to reconstruct and compress consecutive superposition steps, in the style of Sledgehammer's structured proof

© The Author(s) 2026
A. Biere et al. (Eds.): IJCAR 2026, LNAI 16688, pp. 1–18, 2026.
https://doi.org/10.1007/978-3-032-32589-1_1

reconstruction [5, Sect. 6.3]. This would yield a shorter and more high-level proof, in which each step may combine multiple rewrites. Our approach is orthogonal. Our working hypothesis is that the 62-step Vampire proof, which emerged as the byproduct of a saturation process, is likely suboptimal. By mixing and matching proofs generated by different automatic provers, as proposed by Sutcliffe et al. [29], we hoped to achieve a shorter, simpler proof.

We introduce Krympa, a tool that minimizes equational proofs by decomposing them into independently provable parts and reassembling them into more concise, intelligible proofs. Specifically, starting from a Vampire-generated proof, the tool transforms it into a direct proof (Sect. 3) and analyzes its inferences to break it down into intermediate results that serve as candidate lemmas. Each of these lemmas is then proved independently using Vampire and Twee (Sect. 4), the two leading systems in the unit equality division of the 2025 edition of CASC [28]. The resulting proofs are then combined into a single proof using heuristics that favor shorter derivations (Sect. 5).

Given the 62-step Vampire proof of $650 \implies 448$, our tool produces a 20-step proof, where 13 steps are generated by Twee and 7 steps are generated by Vampire (Sect. 6). In a larger empirical evaluation, we applied the tool to 1431 provable implications from the Equational Theories Project and obtained positive results (Sect. 7). In particular, the tool reduced a 151-step Vampire proof to 10 steps.

Our tool is implemented in Rust, OCaml, and Python. Its source code is available at https://github.com/kondylidou/Krympa. The files associated with Tao's challenge and our empirical evaluation data are also available online [20].

An extended version of this paper is available online as a technical report at https://arxiv.org/abs/2605.21200.

2 Background

We briefly review the Vampire and Twee provers and their proof formats.

2.1 Vampire and Superposition Proofs

Vampire is a saturation-based theorem prover for first-order logic with equality based on the superposition calculus [3]. It implements highly optimized search strategies and data structures, and integrates techniques such as literal selection, term orders, redundancy elimination, strategy scheduling, and portfolios.

Superposition works on implicitly universally quantified clauses. A preprocessor performs clausification and Skolemization. For example, the axiom $\forall x.\, \mathsf{f}(x) = \mathsf{g}(x)$ is transformed into $\mathsf{f}(x) = \mathsf{g}(x)$, where x is a free variable, and the conjecture $\forall x.\, \mathsf{f}(x) = \mathsf{g}(x)$ is negated and transformed into $\mathsf{f}(\mathsf{sk}) \neq \mathsf{g}(\mathsf{sk})$, where sk is a Skolem constant. The objective is to derive the contradictory clause $\bot$. For the unit equality fragment, the calculus's two relevant inference rules are as follows:

$$\frac{t \neq u}{\bot}\ \text{equality resolution} \qquad \frac{t = t' \qquad s[u] \bowtie s'}{\mu(s[t'] \bowtie s')}\ \text{superposition}$$

The equality resolution rule has one premise, $t \neq u$, one conclusion, $\bot$, and one side condition: that t and u are unifiable. The superposition rule has two premises and one conclusion. The $\bowtie$ symbol denotes either $=$ or $\neq$ throughout the rule. The $=$ and $\neq$ operators are commutative; for example, the premise $t \neq u$ can match the disequation $\mathsf{f}(\mathsf{a}) \neq x$ either as is or as $x \neq \mathsf{f}(\mathsf{a})$. The premises are assumed to have mutually disjoint sets of variables, which can be achieved by renaming. The notation $s[\,]$ represents a term with a hole, the terms $s[u]$ and $s[t']$ are obtained by filling the hole in $s[\,]$ with u and t', and μ is a most general unifier of t and u. For example, the most general unifier of the terms $\mathsf{h}(\mathsf{a}, y)$ and $\mathsf{h}(x, \mathsf{b})$ is $\{x \mapsto \mathsf{a},\ y \mapsto \mathsf{b}\}$; applying it on both terms yields $\mathsf{h}(\mathsf{a}, \mathsf{b})$. Finally, the rule has further side conditions, not shown here, that restrict the search space.

Example 1. A subtle case of the superposition rule arises when both premises are the same clause. In the following, the variable in the second premise has been renamed to avoid a clash:

$$\frac{\mathsf{f}(\mathsf{f}(x)) = \mathsf{g}(x) \qquad \mathsf{f}(\mathsf{f}(x')) = \mathsf{g}(x')}{\mathsf{f}(\mathsf{g}(x)) = \mathsf{g}(\mathsf{f}(x))}\,\text{superposition}$$

This instance is obtained by taking $t := \mathsf{f}(\mathsf{f}(x))$, $t' := \mathsf{g}(x)$, $\bowtie := =$, $s[\,] := \mathsf{f}(\mathsf{f}(\,))$, $u := \mathsf{f}(x')$, $s' := \mathsf{g}(x')$, and $\mu = \{x' \mapsto \mathsf{f}(x)\}$. Applying the unifier μ to both premises yields the equations $\mathsf{f}(\mathsf{f}(x)) = \mathsf{g}(x)$ and $\mathsf{f}(\mathsf{f}(\mathsf{f}(x))) = \mathsf{g}(\mathsf{f}(x))$. The inference replaces the subterm $\mathsf{f}(\mathsf{f}(x))$ in the second equation with $\mathsf{g}(x)$ using the first equation as a left-to-right rewrite rule, and derives the conclusion. ■

Example 2. Vampire implements *parallel superposition*, a variant of the superposition rule in which all subterms that match a term are replaced. For example:

$$\frac{\mathsf{b} = \mathsf{a} \qquad \mathsf{h}(\mathsf{b}, \mathsf{a}, \mathsf{b}) \neq \mathsf{h}(\mathsf{a}, \mathsf{b}, \mathsf{a})}{\mathsf{h}(\mathsf{a}, \mathsf{a}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{a}, \mathsf{a})}\,\text{parallel superposition}$$

■

Superposition proofs are expressed in a linear format. They are refutational and show how to derive $\bot$ from the input axioms and the negated conjecture.

Example 3. The following is a superposition proof from clauses 1–3:

1. $\mathsf{a} = \mathsf{b}$	axiom
2. $\mathsf{f}(x) = x$	axiom
3. $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$	negated conjecture
4. $\mathsf{h}(\mathsf{b}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{b})$	by parallel superposition from 2 and 3
5. $\mathsf{h}(\mathsf{a}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{a})$	by parallel superposition from 1 and 4
6. $\bot$	by equality resolution from 5

■

2.2 Twee and Structured Equational Chain Proofs

Twee is an automatic prover specialized for equational reasoning. It is based on the unfailing completion procedure [2], an extension of Knuth–Bendix completion

[19]. As with superposition, quantifiers are eliminated by a preprocessor. In the DISCOUNT [10] and Waldmeister tradition [7], Twee's proofs are structured as a sequence of lemmas, where the lemmas and the conjecture are proved by chains of equalities. Twee introduces lemmas if they are needed more than once.

Example 4. The following is a Twee-style proof of goal 1 from axioms 1 and 2:

Axiom 1: $\mathsf{a} = \mathsf{b}$

Axiom 2: $\mathsf{f}(x) = x$

Lemma 3: $\mathsf{f}(\mathsf{b}) = \mathsf{a}$
Proof:
$\mathsf{f}(\mathsf{b})$
$=$ { by axiom 1 right-to-left }
$\mathsf{f}(\mathsf{a})$
$=$ { by axiom 2 }
a

Goal 1: $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$
Proof:
$\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a})$
$=$ { by lemma 3 }
$\mathsf{h}(\mathsf{a}, \mathsf{a})$
$=$ { by lemma 3 right-to-left }
$\mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$

■

3 Conversion to Direct Proofs

Vampire generates proofs by refutation, whereas our mix-and-match approach requires direct proofs. To bridge this gap, we transform Vampire proofs into direct proofs. In the following sections, we will always use direct proofs.

In equational logic, to produce a direct proof, we first introduce existential quantifiers for Skolem constants and universal quantifiers for variables. For example, $\mathsf{h}(x, \mathsf{sk}) \neq x$ is transformed into $\exists z.\, \forall x.\, \mathsf{h}(x, z) \neq x$. Then we apply the contrapositive to all inferences in which a premise and the conclusion are disequations to obtain positive equations. Thus, the inference

$$\frac{\mathsf{h}(\mathsf{a}, y) = \mathsf{b} \qquad \mathsf{h}(x, \mathsf{sk}) \neq x}{\mathsf{b} \neq \mathsf{a}}\ \text{superposition}$$

becomes

$$\frac{\forall y.\, \mathsf{h}(\mathsf{a}, y) = \mathsf{b} \qquad \mathsf{b} = \mathsf{a}}{\forall z.\, \exists x.\, \mathsf{h}(x, z) = x}$$

Notice that negating $\exists z.\, \forall x.\, \mathsf{h}(x, z) \neq x$ flips the quantifiers.

Equality resolution inferences from a premise $t \neq t$ are omitted since their contrapositives derive tautologies $t = t$. This reduces proof length by one.

Example 5. From Example 3's proof by refutation, we obtain this direct proof:

1. $\mathsf{a} = \mathsf{b}$	axiom
2. $\forall x.\, \mathsf{f}(x) = x$	axiom
3. $\mathsf{h}(\mathsf{b}, \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{b})$	from 1 and the tautology $\mathsf{h}(\mathsf{a}, \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{a})$
4. $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$	from 2 and 3

■

We will refer to the statements of proof steps such as 3 and 4 above as *lemmas*. We will say that a lemma ℓ is a *direct dependency* of a lemma ℓ' if ℓ is used in the proof of ℓ', and that ℓ is an *indirect dependency*, or simply *dependency*, of ℓ' if it is transitively a dependency of ℓ'.

4 Proof Generation for Lemmas

Our approach starts by translating the main theorem into a TPTP [13] input problem and running Vampire to produce a baseline proof. This proof is turned into a direct proof. For each lemma in the direct proof, we generate corresponding problems, with the objective of proving them using Vampire and Twee. Three problem variants are generated:

1. *Big-step problems* contain the axioms together with the lemma as the conjecture, and nothing else. This allows us to investigate whether a radically new proof, with different intermediate steps, can be found.
2. *Small-step problems* contain the axioms together with the lemma as the conjecture, and all lemmas derived prior to this lemma in the baseline proof as additional axioms. This allows us to investigate whether a somewhat similar variant of the original derivation can be found.
3. *Abstracted problems* are variants of big-step problems that contain the axioms together with an abstracted, possibly unprovable version of the lemma as the conjecture. Specifically, selected subterms of the lemma—for example, terms such as $x \diamond y$ that do not contain nested applications—are replaced by fresh variables. This allows us to investigate whether a more general version of the lemma is provable, ideally with a shorter, more abstract proof.

Each problem is submitted to the two provers. If a proof is found for a small-step problem, we expand it to recursively include the shortest proofs of the lemmas used as axioms for the axioms referenced in the proof. Ties are broken arbitrarily.

Next, we compare the proofs of the three problem variants corresponding to the same lemma. If the abstracted problem has the shortest proof, the lemma it proves is replaced in all small-step problems where it appears as an axiom with the generalized lemma from the abstracted problem. Each updated small-step problem is then re-proved, and if the result has fewer steps, we replace the small-step problem's proof with it.

The length of a Vampire-generated proof is the number of steps of its direct proof, excluding preprocessing. For Twee, the length of a proof is the cumulative number of equalities in the equality chains. Thus, the Vampire proof in Example 5 has two steps, and the Twee proof in Example 4 has four steps.

5 Proof Construction for the Main Theorem

Based on the lemmas' proofs generated in the previous phase, our approach constructs a proof of the main theorem. The proof generally consists of three segments. The first segment starts with the axioms and ends with the derivation of a so-called *departure lemma*. The second segment derives a so-called *arrival lemma*. The third segment derives the conjecture. Different candidates are considered as the departure and arrival lemmas, yielding different proofs. The proof with the fewest steps is chosen.

Specifically, we first identify up to six lemmas that arise close to the end of the baseline proof, including the conjecture, and consider them as candidate

arrival lemmas. For each candidate arrival lemma, we consider its dependencies as candidate departure lemmas. Then, for each candidate departure lemma, we construct a problem with the axioms and the departure lemma's dependencies as the axioms and the departure lemma itself as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the first segment, unless an even shorter proof was generated in the previous phase.

Next, for each pair of candidate departure and arrival lemmas, we generate a new problem with the original axioms, the departure lemma, and its dependencies as axioms and the arrival lemma as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the second segment, unless an even shorter proof was generated earlier. Finally, we generate a new problem with the original axioms, the departure lemma, its dependencies, and the arrival lemma as axioms and the original conjecture as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the third segment, unless an even shorter proof was generated earlier.

Without the segmentation, proof minimization could be intractable due to combinatorial explosion. We chose to work with three segments, and six candidate arrival lemmas, as a trade-off between performance and flexibility. The number of candidate arrival lemmas is a parameter that can easily be adjusted.

Example 6. Before we review the three-segment proof construction approach in detail, let us look at an example. The following sketch represents a baseline seven-step Vampire direct proof of a theorem $A \Longrightarrow C$, where A denotes the axiom and $L_1, \ldots, L_6$ are the lemmas used to derive the conjecture C:

A	axiom
L_1	from A
L_2	from A and L_1
L_3	from L_1 and L_2
L_4	from L_2 and L_3
L_5	from L_3 and L_4
L_6	from A and L_5
C	from L_5 and L_6

In the first phase, for each lemma $L_1, \ldots, L_6$ and the conjecture C, we construct big-step, small-step, and abstracted problems and try to prove them using Vampire and Twee, retaining the shortest proof for each lemma. Suppose the following: The shortest proof of L_1 has one step and is obtained from its big-step problem using Vampire; for L_2 and L_3, the shortest proofs are obtained from their small-step problems using Twee; for L_4, the shortest proof is obtained from its abstracted problem using Twee; and for L_5, L_6, and C, the shortest proofs are obtained from their small-step problems using Vampire.

In the next phase, the last five lemmas, $L_2, \ldots, L_6$, and the conjecture C are considered as candidate arrival lemmas. We focus on L_6. The proof below, found by Vampire for L_6's small-step problem, is the shortest proof for L_6:

A	axiom
L_1	from A

L_2	from A and L_1
L_3	from L_1 and L_2
L_4	from L_2 and L_3
L_5	from L_3 and L_4
L_6	from A and L_5

This proof happens to be identical to the first six steps of the baseline proof, but in general it could be different.

Next, lemmas L_1 to L_5 are considered as candidate departure lemmas. We focus on L_3. The proof of conjecture C is constructed by concatenating three segments. For the first segment, we create a new problem with A, L_1, and L_2 as axioms, since they are dependencies of the departure lemma L_3 in the above proof of L_6, and L_3 as the conjecture. We run both provers on this problem and obtain a two-step Vampire proof of L_3 from A, L_1, and a new lemma L_2'. Since L_1 is treated as an axiom, we must include its proof to obtain a complete proof of L_3. In the first phase, we found a one-step Vampire proof of L_1 from the axiom A, so we use it. In summary, the proofs of L_1 and L_3 form the first segment, which consists of one step for L_1 and two steps for L_3.

For the second segment, we create a new problem with A, L_1, L_2', and L_3 as axioms and the arrival lemma L_6 as the conjecture. We run both provers on this problem and obtain a two-step Twee proof of L_6 from L_1 and L_3. Together with the first segment, this yields a five-step proof of L_6. Since this proof is shorter than the six-step proof of L_6 presented above, it is used as the second segment.

For the third segment, we create a new problem with A, L_1, L_2', the departure lemma L_3, and the arrival lemma L_6 as axioms and C as the conjecture. We run both provers on this problem and obtain a two-step Twee proof of C from L_2' and L_3. Since this proof does not use the arrival lemma L_6, the second segment is excluded from the result. Concatenating the first and third segments yields a new five-step proof of C, which is two steps shorter than the baseline proof:

A	axiom
L_1	from A
L_2'	from A and L_1
L_3	from L_1 and L_2'
C	by a two-step equality chain using L_2' and L_3

Finally, other combinations of candidate departure and arrival lemmas are considered, and the shortest proof is retained. ■

5.1 Construction of the Dependency Graph

We identify lemmas occurring close to the end of the derivation as candidate arrival lemmas. Different candidates typically depend on substantially different subsets of earlier lemmas. Each candidate therefore induces its own dependency chain, and different choices can lead to substantially different proof lengths. We consider six candidate arrival lemmas extracted from the baseline proof, including the conjecture itself, since our approach may produce a shorter proof

of the conjecture by reproving it directly from a minimized dependency set. All six candidates can be processed in parallel.

For every candidate, we build a dependency graph that captures the lemmas required to derive it. Direct dependencies are determined from the shortest Vampire or Twee proof obtained for each lemma. Here, shortest means fewest inference steps, not fewest axioms. Given that we generate three problem variants and run two provers, up to six proofs per lemma are considered. A lemma ℓ is considered to directly depend on a lemma ℓ' if the shortest proof of ℓ uses ℓ' as an axiom. Thus, for big-step and abstracted problems, only the original axioms can be dependencies. For small-step problems, each step in a Vampire proof and each lemma in a Twee proof is considered a lemma.

The dependency graph associated with a candidate arrival lemma is a directed acyclic graph (DAG) whose nodes correspond to lemmas and whose edges express derivability between them. Formally, let V be a finite set of lemmas, each represented by an equation and a set of direct dependencies on other lemmas. We construct a DAG (V, E), where each vertex $\ell \in V$ corresponds to a lemma and each edge $(\ell, \ell') \in E$ indicates a direct dependency of ℓ on ℓ'. As an optimization, we merge lemmas that are identical up to the naming of variables, keeping the shortest proof.

5.2 Construction of the First Proof Segment

For each candidate arrival lemma, we investigate whether all lemmas included in its dependency graph are needed to derive it or whether a shorter proof can be obtained by choosing a departure lemma and recomputing parts of the derivation by combining proofs generated by the provers.

As candidate departure lemmas, we consider all lemmas in the DAG. Let ℓ be a candidate departure lemma. If ℓ depends only on the axioms, we take the shortest big-step, small-step, or abstracted proof previously found by Vampire or Twee. Otherwise, we build a problem that includes ℓ's dependencies in the DAG as axioms and the departure lemma as the conjecture, and we run Vampire and Twee. If at least one of them succeeds, we choose the shorter proof as ℓ's proof. This derivation, together with the shortest proofs of ℓ's dependencies generated for the big-step, small-step, or abstracted problems, forms the first segment of the final proof. However, if we found an even shorter proof for the big-step, small-step, or abstracted problem, we use that proof instead. For small-step proofs, we must also include the proofs of the lemmas encoded as axioms.

5.3 Construction of the Remaining Proof Segments

To construct the second segment, we generate a problem with the departure lemma and its dependencies as axioms and the arrival lemma as the conjecture, and run both provers. If at least one of them succeeds, we choose the shorter proof as the proof of the arrival lemma. As above, we fall back on the proof of a big-step, small-step, or abstracted problem if it is even shorter.

Finally, to construct the third segment, we generate a problem with the departure lemma, the arrival lemma, and their dependencies as axioms and the original conjecture as the conjecture, and invoke both provers. If at least one of them succeeds, we choose the shorter proof as the proof of the original conjecture. As above, we fall back on a previously derived proof if it is even shorter.

The final proof is obtained by concatenating the three segments. When we concatenate, lemmas from one segment are passed as axioms to subsequent segments. Twee equality chains used to prove lemmas are not visible in subsequent segments. The final proof might contain unreferenced lemmas; these are pruned.

5.4 Proof Output

Our tool generates the minimized proof in a native format, from which two Lean outputs are produced. The first Lean output is a step-by-step formalization using the `calc` tactic to reconstruct equality chains. It applies the `duper` tactic to fill in the subproofs. For example, a proof of $t_1 = t_2 = t_3 = t_4$ becomes

```
calc
  t₁ = t₂ := by duper ...
  _ = t₃ := by duper ...
  _ = t₄ := by duper ...
```

where the ellipses stand for `duper`'s arguments. The second Lean output is a more compact formalization in which each lemma is proved directly using Lean's automation without including the intermediate steps in equality chains.

6 Application to Tao's Challenge

We implemented our approach and tried the resulting tool, Krympa, on Tao's challenge theorem 650 $\Longrightarrow$ 448:

$$(\forall x, y, z.\ x = x \diamond (y \diamond ((z \diamond x) \diamond y))) \Longrightarrow \forall x, y, z.\ x = x \diamond (y \diamond (z \diamond (x \diamond z))).$$

Our tool first ran Vampire to obtain a baseline 62-step superposition proof. Then it constructed 62 problems of each variant (big-step, small-step, and abstracted) and tried to prove them using Vampire and Twee. Among the six candidate arrival lemmas, the shortest proof was found by selecting

$$\forall x, y, z.\ x = x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x). \qquad \text{(lemma 9)}$$

The coloring highlights repeating patterns. Next, our tool constructed the dependency graph for this lemma. The DAG contained 37 lemmas. It was based on big- and small-step proofs.

Among the 10 candidate departure lemmas, our tool found the shortest proof by selecting

$$\forall x, y, z, w.\ (x \diamond ((y \diamond x) \diamond x)) \diamond z = ((x \diamond ((y \diamond x) \diamond x)) \diamond z) \diamond (w \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond w)). \qquad \text{(lemma 7)}$$

According to the DAG, the shortest proof of this lemma was found by running Vampire on the small-step problem consisting of the axiom and

$$\forall x, y, z, w.\ x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x) = (x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x)) \diamond (w \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond w)). \quad \text{(lemma 5)}$$

The shortest proof of lemma 5 was found by running Vampire on the corresponding big-step problem. The following lemmas were derived:

$$\forall x, y, z, w.\ x \diamond ((y \diamond z) \diamond x) = (x \diamond ((y \diamond z) \diamond x)) \diamond (w \diamond (z \diamond w)) \quad \text{(lemma 1)}$$

$$\forall x, y, z, w, v, u.\ x \diamond ((y \diamond ((z \diamond w) \diamond y)) \diamond x) = (x \diamond ((y \diamond ((z \diamond w) \diamond y)) \diamond x)) \diamond (v \diamond ((u \diamond (w \diamond u)) \diamond v)) \quad \text{(lemma 2)}$$

$$\forall x, y, z, w, v.\ x \diamond (y \diamond x) = (x \diamond (y \diamond x)) \diamond (z \diamond ((w \diamond ((v \diamond y) \diamond w)) \diamond z)) \quad \text{(lemma 3)}$$

$$\forall x, y, z, w, v.\ x \diamond (y \diamond x) = (x \diamond (y \diamond x)) \diamond ((z \diamond (y \diamond z)) \diamond (w \diamond ((v \diamond y) \diamond w))) \quad \text{(lemma 4)}$$

Specifically, following the inference steps of the baseline Vampire proof, our tool derived lemma 1 by applying a superposition inference with the axiom $x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ as the first premise and a renamed copy $x' = x' \diamond (y' \diamond ((z' \diamond x') \diamond y'))$ as the second premise. The most general unifier of the first premise's right-hand side and the subterm $z' \diamond x'$ of the second premise is $\{x' \mapsto y \diamond ((z \diamond x) \diamond y),\ z' \mapsto x\}$. Applying the unifier to both premises yields the equations $x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ and $y \diamond ((z \diamond x) \diamond y) = (y \diamond ((z \diamond x) \diamond y)) \diamond (y' \diamond ((x \diamond (y \diamond ((z \diamond x) \diamond y))) \diamond y'))$. The superposition inference replaced the subterm $x \diamond (y \diamond ((z \diamond x) \diamond y))$ in the second equation with x using the first equation as a right-to-left rewrite rule, and thus derived lemma 1, up to the naming of variables. Lemmas 2 to 4 were derived similarly following the steps of the baseline Vampire proof.

Next, from the axiom and lemma 5 our tool proved the departure lemma (lemma 7) using Vampire. In the proof,

$$\forall x, y, z, w.\ (x \diamond ((y \diamond x) \diamond x)) \diamond z = ((x \diamond ((y \diamond x) \diamond x)) \diamond z) \diamond ((w \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond w)) \diamond (z \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond z))) \quad \text{(lemma 6)}$$

was derived by applying a superposition inference with the axiom as the first premise and lemma 5 as the second premise. Then, lemma 7 was derived by applying a superposition inference with lemma 6 as the first premise and lemma 5 as the second premise.

After, from the axiom and lemma 7, our tool proved the arrival lemma (lemma 9) using Twee. For this proof, Twee introduced the intermediate step

$$\forall x, y, z, w.\ (y \diamond ((z \diamond y) \diamond y)) \diamond w = ((y \diamond ((z \diamond y) \diamond y)) \diamond w) \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x). \quad \text{(lemma 8)}$$

Finally, assuming all the lemmas derived so far, our tool proved the conjecture from lemmas 5 and 9 using Twee. The resulting proof has 20 steps, including three Twee-generated chains of equalities.

Below we present the final proof adapted from our tool's detailed Lean output. Instead of relying on proof automation, we use the `nth_rw` tactic, which performs a single rewrite step, where the numeric argument indicates which matching occurrence should be rewritten. In one case, two numbers are supplied, corresponding to a parallel rewrite.

```
theorem Equation650_implies_Equation448 (G : Type _) [Magma G]
      (op_law : ∀ x y z : G, x = x ◇ (y ◇ ((z ◇ x) ◇ y))) :
    ∀ x y z : G, x = x ◇ (y ◇ (z ◇ (x ◇ z))) :=
  have lemma1 (x y z w : G) :
      x ◇ ((y ◇ z) ◇ x) = (x ◇ ((y ◇ z) ◇ x)) ◇ (w ◇ (z ◇ w)) := by
    nth_rw 3 [op_law z x y]
    exact op_law (x ◇ ((y ◇ z) ◇ x)) w z

  have lemma2 (x y z w v u : G) :
      x ◇ ((y ◇ ((z ◇ w) ◇ y)) ◇ x) =
      (x ◇ ((y ◇ ((z ◇ w) ◇ y)) ◇ x)) ◇ (v ◇ ((u ◇ (w ◇ u)) ◇ v)) := by
    nth_rw 1 2 [lemma1 y z w u]
    exact lemma1 x (y ◇ ((z ◇ w) ◇ y)) (u ◇ (w ◇ u)) v

  have lemma3 (x y z w v : G) :
      x ◇ (y ◇ x) = (x ◇ (y ◇ x)) ◇ (z ◇ ((w ◇ ((v ◇ y) ◇ w)) ◇ z)) := by
    nth_rw 1 [lemma1 w v y x]
    exact op_law (x ◇ (y ◇ x)) z (w ◇ ((v ◇ y) ◇ w))

  have lemma4 (x y z w v : G) :
      x ◇ (y ◇ x) = (x ◇ (y ◇ x)) ◇ ((z ◇ (y ◇ z)) ◇ (w ◇ ((v ◇ y) ◇ w))) := by
    nth_rw 1 [lemma1 w v y z]
    exact lemma3 x y (z ◇ (y ◇ z)) w v

  have lemma5 (x y z w : G) :
      x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x) =
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) ◇ (w ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ w)) := by
    nth_rw 1 [lemma2 w y z y x ((z ◇ y) ◇ y)]
    exact lemma4 x (y ◇ ((z ◇ y) ◇ y)) w x ((z ◇ y) ◇ y)

  have lemma6 (x y z w : G) :
      (x ◇ ((y ◇ x) ◇ x)) ◇ z =
      ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ ((w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) ◇
        (z ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ z))) := by
    nth_rw 1 [lemma5 z x y w]
    exact op_law ((x ◇ ((y ◇ x) ◇ x)) ◇ z) (w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) z

  have lemma7 (x y z w : G) :
      (x ◇ ((y ◇ x) ◇ x)) ◇ z =
      ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ (w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) := by
    nth_rw 1 [lemma5 w x y z]
    exact lemma6 x y z w

  have lemma8 (x y z w : G) :
```

```
    ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w) =
    (x ◇ ((y ◇ x) ◇ x)) ◇ z := by
    let T := x ◇ ((y ◇ x) ◇ x)
    calc
      (T ◇ z) ◇ (T ◇ w) =
      ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇ (T ◇ (T ◇ w)))))) := by
        nth_rw 1 [←op_law]
      _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇
            (T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w))))))))  := by
        nth_rw 1 [←lemma7]
      _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇
            ((T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w)))) ◇ (((T ◇ w) ◇ (w ◇ (T ◇ w))) ◇
              (T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w))))))))))  := by
        nth_rw 2 [←lemma7]
      _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ (w ◇ (T ◇ w))))) := by
        nth_rw 1 [←op_law]
      _ = ((T ◇ z) ◇ ((T ◇ w) ◇ (T ◇ (T ◇ w)))) := by
        nth_rw 1 [←lemma7]
      _ = ((x ◇ ((y ◇ x) ◇ x)) ◇ z) := by
        nth_rw 1 [←lemma7]

  have lemma9 (x y z : G) :
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) = x := by
    calc
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) =
      (x ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x))) := by
        nth_rw 1 [lemma8]
      _ = (x ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇
            ((y ◇ ((z ◇ y) ◇ y)) ◇ x)))) := by
        nth_rw 2 [lemma8]
      _ = x := by
        nth_rw 1 [←op_law]

  show _ by
    intros x y z
    calc
      x = x ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ x) := by
        nth_rw 1 [lemma9]
      _ = (x ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ x)) ◇ ((y ◇ (z ◇ (x ◇ z))) ◇
            ((x ◇ ((y ◇ x) ◇ x)) ◇ (y ◇ (z ◇ (x ◇ z))))) := by
        nth_rw 1 [←lemma5]
      _ = x ◇ ((y ◇ (z ◇ (x ◇ z))) ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇
            (y ◇ (z ◇ (x ◇ z))))) := by
        nth_rw 1 [lemma9]
      _ = x ◇ (y ◇ (z ◇ (x ◇ z))) := by
        nth_rw 1 [lemma9]
```

7 Experiments on Other Equational Proofs

To assess the general potential of our approach, we evaluated our tool on a set of equational theorems obtained from the Equational Theories Project repository [6]. We selected all problems in the 13 Lean files `Proofs1` to `Proofs13` that have a proof and translated them to TPTP problem files, yielding 1431 benchmarks.

For each file, we invoked our tool's TPTP problem generator, which parses the Lean theorems and produces corresponding TPTP problem files. For each problem, our tool was given 600 seconds to produce a minimized proof using Vampire and Twee; on failure, the baseline Vampire proof was output. A time limit of 10 seconds was used for each prover invocation. The 600-second overall limit is generous; most proofs are minimized in well under a minute. The experiments were conducted on a server equipped with a dual-socket AMD EPYC 9965 system (384 cores, 768 threads) running at 2.25–3.70 GHz with 3 TiB of DDR5 ECC RAM, and running Debian GNU/Linux 13 (kernel 6.17.13+deb13-amd64).

Overall, proofs for the 13 Lean files have an average length of 6.6 steps before minimization and 4.5 steps after minimization using the combination of small-step and abstracted problems and both provers. This corresponds to a 31.5% decrease, showing that even short proofs can often be made shorter.

Since longer proofs present more opportunities for minimization, we now focus on the 117 benchmarks whose baseline proofs have at least 15 steps. Table 1 compares proof lengths before and after minimization. The "Avg. before" column shows the average number of steps in the baseline proofs. The "Avg. after" columns report the average proof length after minimization under four configurations, which differ in which problem variants are used: "BA" denotes the

Table 1. Comparison of proof lengths before and after minimization for problems with baseline proofs of at least 15 steps

File	Num. problems	Avg. before	Avg. after			
			BA	SA	BS	BSA
`Proofs1`	9	17.3	16.0	**13.1**	13.3	13.3
`Proofs2`	8	16.9	14.4	**11.5**	**11.5**	**11.5**
`Proofs3`	7	19.3	15.6	10.9	**10.7**	10.9
`Proofs4`	11	19.1	14.3	**10.9**	11.4	11.4
`Proofs5`	9	20.1	17.6	12.8	**11.9**	**11.9**
`Proofs6`	11	25.6	18.9	**12.5**	12.6	12.6
`Proofs7`	11	37.2	19.7	**11.8**	11.9	11.9
`Proofs8`	7	24.4	15.6	**12.3**	13.1	13.1
`Proofs9`	14	39.8	29.0	**13.1**	14.1	14.1
`Proofs10`	2	21.5	16.0	**8.0**	11.0	11.0
`Proofs11`	5	25.4	22.4	**13.0**	14.0	14.0
`Proofs12`	13	24.6	16.5	**8.0**	8.5	8.5
`Proofs13`	10	35.3	27.7	**9.1**	10.1	10.1
Total	117	26.3	19.2	**11.4**	11.8	11.8

combination of the big-step and abstracted variants; "SA" denotes the combination of the small-step and abstracted variants; "BS" denotes the combination of the big- and small-step variants; and "BSA" denotes the combination of all three variants.

The results show an often substantial reduction in proof length. SA generally yielded the shortest proofs. Across all problems for the 13 Lean files, the average reduction with SA is 56.7%. BS and BSA also produced substantial reductions, whereas BA generally yielded the least improvements.

It might seem counterintuitive that SA, which does not consider big-step problems, outperforms BSA. However, the nonmonotonicity is to be expected. Provers are nondeterministic, especially when invoked with a time limit; for example, a prover might sometimes find a proof after 9.98 seconds and sometimes time out after 10 seconds. Moreover, our approach makes different heuristic choices when constructing the three proof segments depending on which problem variants are used. As a result, SA might find a short proof that escapes BSA.

The reduction in proof length is especially noticeable in individual cases. The problem 2666 $\Longrightarrow$ 3460 has a Vampire proof with 51 inference steps, which our tool reduces to only 12 single rewrite steps, and 2923 $\Longrightarrow$ 2628 is reduced from 180 steps to only 34. The problem 3569 $\Longrightarrow$ 3957 is reduced from 92 to 23 steps and, even more dramatically, 3957 $\Longrightarrow$ 3971 is reduced from 141 steps to only 23. Furthermore, 2860 $\Longrightarrow$ 2660 is reduced from 44 to 14 steps, and 723 $\Longrightarrow$ 872 goes from 57 to 13 steps. Finally, 947 $\Longrightarrow$ 3897 underwent the largest reduction, from 151 to 10 steps. Overall, these results demonstrate that our approach produces shorter proofs across a diverse set of equational theorems.

8 Related Work

At least two other researchers took on Tao's challenge. Kinyon [18] found a 24-step proof (excluding preprocessing) of 650 $\Longrightarrow$ $\forall x, y.\ x = x \diamond y$ using Prover9 [22], from which 650 $\Longrightarrow$ 448 follows by instantiation. Later, Le Floch [11] developed a pen-and-paper proof and translated it to Lean. The Lean proof relies on only 14 rewrite steps but includes additional reasoning as proof terms, and two of the rewrite steps are parallel, so the overall length is similar to ours. The proof is "loosely based" on the output of multiple Prover9 runs "with intermediate results thrown in as assumptions or as goals"—in essence, a manual approximation of our approach. Also in the context of the Equational Theories Project, Janota [16] evaluated Vampire on the project's problems and showed that combining superposition with finite model finding can solve almost all problems. One reason for this success might be the work on term ordering diagrams by Hajdu et al. [15].

We are aware of little work on automated proof minimization. Stachniak [27] designed an algorithm for constructing resolution proofs in propositional logics known as strongly finite logics. Amjad [1] and Cotton [9] introduced techniques for minimizing propositional resolution proofs. Vyskočil et al. [31] proposed to compress proofs by inventing new definitions using a heuristic based

on substitution trees. Gu et al. [14] developed ProofOptimizer, which uses large language models to simplify Lean proofs.

Some SAT (satisfiability) and SMT (satisfiability modulo theories) solvers can minimize the number of axioms needed for a proof, but the result can be a longer proof. SAT solvers commonly interleave search, which can be expressed as resolution steps, with formula-rewriting techniques that go beyond resolution. This interleaving, known as inprocessing [17], is highly effective and often yields both faster solving times and shorter proofs than either approach in isolation.

The idea of automatically mixing and matching proofs is not new. Sutcliffe et al. [29] introduced a method for combining automatically generated proofs to generate new ones. Their proofs are represented as DAGs, enabling the identification and replacement of subproofs across different proofs. Proof combination is guided by heuristics that measure structural similarity, and a greedy search strategy is used to explore alternative combinations that yield proofs differing from the originals. In contrast to our approach, the main objective is to increase proof diversity rather than minimize proof length.

9 Conclusion

Historically, more research has gone into finding proofs automatically than into improving and presenting them. We introduced an approach for minimizing equational proofs by mixing and matching the output of separate runs of Vampire and Twee, and implemented it in a new tool, Krympa. We used the tool to minimize the proof of problem 650 $\Longrightarrow$ 448 from the Equational Theories Project from 62 to 20 steps, thereby providing a fully automatic solution to a challenge posed by Tao. We also obtained remarkable reductions on other problems originating from the project. The shorter proofs are arguably easier to understand by humans and sometimes more general. Our work shows that proof automation and readability can go hand in hand.

Our approach could be extended in several ways. First, it could be generalized to support full first- or higher-order logic. Second, alternative lemma abstraction strategies could be explored. Third, proofs with four or more segments could be synthesized. Fourth, we might want to consider not only the number of steps but also term size when measuring proofs, as suggested by Le Floch [12]. Fifth, we could try to translate Vampire proofs to Twee's structured equality chain format.

Some possible extensions specifically concern the implementation. First, we could explore whether nondefault Vampire and Twee strategies can produce shorter proofs, following private suggestions by Martin Suda and Nicholas Smallbone. Second, since proof generation relies heavily on external provers, performance could benefit from better scheduling of prover invocations, using adaptive time limits. Finally, our approach is not tied to Vampire and Twee specifically. Vampire is representative of ATP systems that produce TSTP output, such as E and iProver, whereas Twee is representative of completion-based provers such as Waldmeister. Together, they cover most equational provers. Integrating further provers should be straightforward.

Acknowledgments. We thank Bruno Le Floch, Andrew Reynolds, Stephan Schulz, and Uwe Waldmann for fruitful discussions. We thank Laura Kovács, Luca Maio, Martin Suda, Mark Summerfield, and the anonymous reviewers for helpful textual suggestions.

Blanchette's research was cofunded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Heule's research is supported by the NSF under grant DMS-2434625 and funding from AFRL and DARPA under Agreement FA8750-24-9-1000.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this paper.

References

1. Amjad, H.: Compressing propositional refutations. In: Merz, S., Nipkow, T. (eds.) AVoCS 2006. Electronic Notes in Theoretical Computer Science, vol. 185, pp. 3–15. Elsevier (2006)
2. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Aït-Kaci, H., Nivat, M. (eds.) Rewriting Techniques, pp. 1–30. Academic Press (1989)
3. Bachmair, L., Ganzinger, H.: Strict basic superposition. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS, vol. 1421, pp. 160–174. Springer (1998)
4. Bártek, F., Bhayat, A., Coutelier, R., Hajdú, M., Hetzenberger, M., Hozzová, P., Kovács, L., Rath, J., Rawson, M., Reger, G., Suda, M., Schoisswohl, J., Voronkov, A.: The Vampire diary. In: Piskac, R., Rakamaric, Z. (eds.) CAV 2025. LNCS, vol. 15933, pp. 57–71. Springer (2025)
5. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Automated Reas. **56**, 155–200 (2016)
6. Bolan, M., Breitner, J., Brox, J., Carlini, N., Carneiro, M., van Doorn, F., Dvorak, M., Goens, A., Hill, A., Husum, H., Mejia, H.I., Kocsis, Z.A., Floch, B.L., Bar-on, A.L., Luccioli, L., McNeil, D., Meiburg, A., Monticone, P., Nielsen, P., Osazuwa, E.O., Paolini, G., Petracci, M., Reinke, B., Renshaw, D., Rossel, M., Roux, C., Scanvic, J., Srinivas, S., Tadipatri, A.R., Tao, T., Tsyrklevich, V., Vaquerizo-Villar, F., Weber, D., Zheng, F.: The Equational Theories Project (2025)
7. Buch, A., Hillenbrand, T.: WALDMEISTER: Development of a high performance completion-based theorem prover. Tech. rep. (1996)
8. Clune, J., Qian, Y., Bentkamp, A., Avigad, J.: Duper: A proof-producing superposition theorem prover for dependent type theory. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) ITP 2024. LIPIcs, vol. 309, pp. 1–20. Leibniz-Zentrum für Informatik (2024)
9. Cotton, S.: Two techniques for minimizing resolution proofs. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 306–312. Springer (2010)
10. Denzinger, J., Schulz, S.: Recording, analyzing and presenting distributed deduction processes. In: Hong, H. (ed.) PASCO 1994. Lecture Notes Series in Computing, vol. 5, pp. 114–123. World Scientific (1994)

11. Floch, B.L.: Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)". Zulip (2025), https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518970125
12. Floch, B.L.: Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)". Zulip (2025), https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/568313777
13. Geoff Sutcliffe: Stepping stones in the TPTP World. In: Benzmüller, C., Heule, M., Schmidt, R. (eds.) IJCAR 2024. pp. 30–50. LNCS (2024)
14. Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., Markosyan, A.H.: ProofOptimizer: Training language models to simplify proofs without human demonstrations (2025), https://arxiv.org/abs/2510.15700
15. Hajdu, M., Coutelier, R., Kovács, L., Voronkov, A.: Term ordering diagrams. In: Barrett, C.W., Waldmann, U. (eds.) CADE-30. LNCS, Springer, to appear
16. Janota, M.: Experimental results for vampire on the equational theories project (2025), https://arxiv.org/abs/2508.15856
17. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer (2012)
18. Kinyon, M.: Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)". Zulip (2025), https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518961204
19. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
20. Kondylidou, L., Blanchette, J., Heule, M.: Tao's equational proof challenge accepted. Zenodo (2026), https://doi.org/10.5281/zenodo.18624123
21. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: CPP 2023. pp. 253–266. ACM (2023)
22. McCune, W.: Prover9 and Mace4 (2005–2010)
23. Morrison, K., de Moura, L.: `grind`: An SMT-inspired tactic for Lean 4 (short paper—system description). In: Biere, A., Lutz, C., Negri, S. (eds.) IJCAR 2026. LNCS, Springer, to appear
24. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) CADE-28. LNCS, vol. 12699, pp. 625–635. Springer (2021)
25. Norman, C., Avigad, J.: Canonical for automated theorem proving in Lean. In: ITP 2025. LIPIcs, vol. 352, pp. 1–20. Leibniz-Zentrum für Informatik (2025)
26. Smallbone, N.: Twee: An equational theorem prover. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, vol. 12699, pp. 602–613. Springer (2021)
27. Stachniak, Z.: Minimization of resolution proof systems. Fundam. Informaticae **14**(1), 129–146 (1991)
28. Sutcliffe, G.: The 12th IJCAR Automated Theorem Proving System Competition—CASC-J12. AI Communications **38**, 3–20 (2025)
29. Sutcliffe, G., Chang, C., McGuinness, D., Lebo, T., Ding, L., da Silva, P.P.: Combining proofs to form different proofs. In: Fontaine, P., Stump, A. (eds.) PxTP 2011. pp. 60–73. LNCS (2011)

30. Tao, T.: Zulip thread "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)". Zulip (2025), https://leanprover-community.github.io/archive/stream/219941-Machine-Learning-for-Theorem-Proving/
31. Vyskočil, J., Stanovský, D., Urban, J.: Automated proof compression by invention of new definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 447–462. Springer (2010)
32. Zhu, T., Clune, J., Avigad, J., Jiang, A.Q., Welleck, S.: Premise selection for a Lean hammer (2026), https://arxiv.org/abs/2506.07477

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Readable Proofs for the Horn Fragment

Lydia Kondylidou
l.kondylidou@ifi.lmu.de
LMU Munich
Munich, Germany

Jasmin Blanchette
jasmin.blanchette@ifi.lmu.de
LMU Munich
Munich, Germany

Nicholas Smallbone
nicsma@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Uwe Waldmann
uwe@mpi-inf.mpg.de
Max Planck Institute for Informatics
Saarland Informatics Campus
Saarbrücken, Germany

Abstract

Proofs generated by resolution and superposition provers can be difficult for humans to read and understand, partly because the proofs are by contradiction, and partly because they rely on unification. We introduce a structured proof format for the Horn fragment of first-order logic with equality. The format expresses direct proofs. It combines equational rewriting chains and human-style "have/hence" derivations. It relies on matching but not full unification, increasing intelligibility. To interface with provers, we introduce an algorithm that translates resolution and superposition proofs to our format and have implemented it in a tool called Tälja. We have also connected Tälja to the Twee equational prover, which supports the Horn fragment via an encoding. We have evaluated our approach on TPTP benchmarks.

Keywords: First-order logic, Horn fragment, resolution, superposition, proofs

1 Introduction

Automatic theorem provers are useful tools, but the proofs they produce are often cryptic. A computer can check that a sequence of inference steps is correct, but a human trying to understand *why* a theorem holds gains little from a long list of clauses derived using low-level inference rules. This is unfortunate. Users of automatic tools often value not only the final QED but also any insights that a proof might convey. Moreover, when an automatic prover is used inside a proof assistant, the generated proofs are translated and embedded into a formal development [1, 10], so they ought to be readable.

Proofs found by resolution provers such as PyRes [33] and superposition provers such as E [32] and Vampire [24] can be particularly difficult to read and understand. The superposition calculus [6] generalizes the resolution calculus [31] with equality reasoning. For both calculi, two characteristics come together to make the proofs unintelligible. First, the proofs are *by contradiction*: The prover negates the goal to be proved, adds the negated goal to the set of axioms, and then derives falsity. To understand such a proof, we must keep track of those steps that depend on the negated goal and those that do not, a cognitive burden that grows with proof size. Second, to check a proof step, we must try to unify specific syntactic elements. Unification is subtle enough on its own, and in addition, most provers do not indicate the positions at which inferences occur, which must be reconstructed by trial and error. In the following example, it is not obvious whether the premises entail the conclusion:

$$\frac{\mathsf{P}(x, \mathsf{f}(y, y), y, \mathsf{f}(\mathsf{a}, \mathsf{a})) \qquad \neg\mathsf{P}(x, y, y, x) \vee \neg\mathsf{P}(x, x, y, y)}{\neg\mathsf{P}(\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{a})), \mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{a})))}$$

We introduce a structured proof format for the *Horn fragment* of the formalism underlying superposition, first-order logic with equality. Horn clauses form a well-studied, well-behaved fragment; in particular, theorems within the fragment can be proved without leaving the fragment. Our format is intended as a stepping stone toward readable proofs for full first-order logic and higher-order logic [8, 9, 36].

The format aims for human readability in two ways. First, it expresses *direct proofs*, which proceed from the axioms toward the goal, rather than proofs by contradiction. We can follow the reasoning without mentally tracking which intermediate steps originate from the negated goal. Second, proof steps rely on *matching* instead of full unification. Matching is a one-sided form of unification that binds variables in the rewrite rule to subterms of the term being rewritten.

The format is a hybrid. First, it supports equality chains in the style of the DISCOUNT [5], Waldmeister [20], and Twee [35] equational provers. In the purely equational case, a proof consists of a sequence of lemmas, each justified by an equality chain, followed by the restatement of the goal, also justified by an equality chain. An equality chain is a sequence $t_1 \approx t_2 \approx \cdots \approx t_n$ in which each $\approx$ step rewrites a subterm of the preceding term using an axiom or previously proved lemma as a rewrite rule. The equality symbol is written $\approx$ to distinguish it from syntactic equality $=$.



This work is licensed under a Creative Commons Attribution 4.0 International License.

Example 1. The following proof establishes the goal g(g(a)) ≈ a from two axioms via a lemma:

Axiom 1: $g(x) \approx f(x)$

Axiom 2: $f(x) \approx x$

Lemma 3: $g(x) \approx x$
Proof:
  $g(x)$
≈ { by axiom 1 }
  $f(x)$
≈ { by axiom 2 }
  x

Goal 1: g(g(a)) ≈ a
Proof:
  g(g(a))
≈ { by lemma 3 }
  g(a)
≈ { by lemma 3 }
  a ■

Second, our format supports hyperresolution [7] steps. From an implication $A_1 \wedge \cdots \wedge A_m \rightarrow B$ and proofs of instances of the premises, it derives a corresponding instance of the conclusion. This is expressed using a suggestive, Isar-inspired [39] "have/hence" block. A "have" step states an intermediate fact, "hence" derives the conclusion, and multiple premises are combined with "and." Equality chains and "have/hence" blocks can be combined in the same proof.

Example 2. The following hyperresolution-based proof establishes R(a, b) from five axioms via a lemma:

Axiom 1: U(b)

Axiom 2: $U(x) \rightarrow Q(x)$

Axiom 3: S(a)

Axiom 4: $S(x) \rightarrow P(x)$

Axiom 5: $P(x) \wedge Q(y) \rightarrow R(x, y)$

Lemma 6: Q(b)
Proof:
  have U(b)
    by axiom 1
  hence Q(b)
    by axiom 2

Goal 1: R(a, b)
Proof:
  have S(a)
    by axiom 3
  hence P(a)
    by axiom 4
  and Q(b)
    by lemma 6
  hence R(a, b)
    by axiom 5 ■

To interface with existing provers, we introduce a translation algorithm that converts resolution and superposition proofs to our format. The algorithm first expands the given proof, which corresponds to a directed acyclic graph (DAG), into a proof tree. It then processes relevant clauses in order, translating each into a hyperresolution step expressed as a "have/hence" block; equational rewriting steps are assembled into equality chains. We prove the algorithm correct.

We have implemented the algorithm in a tool called *Tälja*. The tool works with proofs generated by E and Vampire from Horn problems. We have also integrated Tälja with Twee, an equational prover that supports the Horn fragment via an encoding. This integration required extending Twee to output resolution and rewriting proofs. We have evaluated Tälja and the Twee integration on relevant categories of the de facto standard TPTP (Thousands of Problems for Theorem Provers) [38] benchmark suite to confirm our approach's usefulness.

2 Background

We briefly describe clausal first-order logic as well as the resolution and superposition calculi based on it. The focus is on the aspects that are relevant to proofs and proof generation.

2.1 Clausal First-Order Logic

We work in simply typed classical first-order logic. We assume a fixed set of *base types* and an infinite set X of typed variables w, x, y, z. A *signature* Σ consists of typed function symbols a, b, c, f, g, ... and typed predicate symbols P, Q, R, ...

Terms s, t, u, v are defined inductively as being either a variable $x \in X$ or the application $f(t_1, \ldots, t_n)$ of an n-ary function symbol f from Σ to n suitably typed terms $t_1, \ldots, t_n$. *Atoms* A, B are applications $P(t_1, \ldots, t_n)$ of an n-ary predicate symbol P from Σ to n suitably typed terms. For both kinds of applications, we omit the parentheses if $n = 0$. *Literals* L are atoms A or negated atoms $\neg A$; we call the former *positive* literals and the latter *negative* literals. Given a literal L, we write $\overline{L}$ for its complement, with $\overline{A} = \neg A$ and $\overline{\neg A} = A$.

Clauses C, D, E are finite multisets $\{L_1, \ldots, L_n\}$ of literals, usually written as $L_1 \vee \cdots \vee L_n$; thus, $P(x) \vee Q(y)$ and $Q(y) \vee P(x)$ are considered syntactically equal. The case where $n = 0$ corresponds to the empty clause, written $\bot$. Variables in clauses are regarded as being implicitly universally quantified; thus, the clause $P(x) \vee Q(y)$ is equivalent to the formula $\forall x, y.\ P(x) \vee Q(y)$.

Horn clauses are clauses that contain at most one positive literal. It is customary to write the Horn clauses

$$\neg A_1 \vee \cdots \vee \neg A_m \vee B \quad \text{and} \quad \neg A_1 \vee \cdots \vee \neg A_m$$

as

$$A_1 \wedge \cdots \wedge A_m \rightarrow B \quad \text{and} \quad A_1 \wedge \cdots \wedge A_m \rightarrow \bot$$

respectively. A special case of the Horn fragment is *equational logic*, in which equality is the only predicate symbol, and all axioms and the goal are single positive literals.

Terms, atoms, literals, and clauses are given a formal semantics by means of an *interpretation* J, which maps types to nonempty sets, function symbols to functions, and predicate symbols to predicates. The equality symbol $\approx$ is always interpreted as meaning equality. The interpretation is complemented by a *variable assignment* ξ. A clause is *valid* if, for all interpretations and variable assignments, one of its literals evaluates to true.

2.2 Resolution Calculus

Resolution is a calculus for clausal first-order logic. It was initially developed by Robinson [31] and has been refined and extended over the years [7]. Our work's primary focus is on superposition, but resolution is a useful stepping stone.

Resolution provers work by saturation: From the axioms and the negated goal, they repeatedly apply inference rules that create conclusions, from which further inferences may apply, until—in case of success—the empty clause $\bot$ is derived. The resolution calculus consists of the inference rule

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\sigma}\ \text{resolution}$$

The substitution σ is a most general unifier (MGU) of A and B. In particular, applying σ to A and B makes them syntactically equal. For example, an MGU of $\mathsf{P}(\mathsf{a}, y)$ and $\mathsf{P}(x, \mathsf{b})$ is $\{x \mapsto \mathsf{a}, y \mapsto \mathsf{b}\}$; applying it to both atoms yields $\mathsf{P}(\mathsf{a}, \mathsf{b})$.

The rule operates on clauses matching the premises and derives new clauses. Since clauses are multisets, a schematic premise such as $D \vee \neg B$ stands for an arbitrary clause that contains a negative literal $\neg B$ anywhere and zero or more literals forming the subclause D. Furthermore, although this is implicit above, variables in the two premises are renamed apart before the rule is applied. An important special case of resolution arises when C and D are $\bot$. Then the conclusion $(C \vee D)\sigma$ is also $\bot$.

The inference rule is *sound*: If the premises are true according to an interpretation J and every variable assignment, then the conclusion is also true according to J and every variable assignment. Resolution proofs show how to derive $\bot$ from the problem's axioms and negated goal, thereby establishing that the goal follows from the axioms. These proofs are conceptually DAGs but usually expressed in a sequential format.

Example 3. A resolution proof from clauses 1–3 follows:

1.	$\mathsf{P}(z', z') \vee \neg\mathsf{Q}(z)$	axiom
2.	$\mathsf{P}(x', \mathsf{b}) \vee \mathsf{Q}(\mathsf{f}(x', x))$	axiom
3.	$\neg\mathsf{P}(\mathsf{a}, y)$	negated goal
4.	$\neg\mathsf{Q}(z)$	by resolution from 1 and 3
5.	$\mathsf{Q}(\mathsf{f}(\mathsf{a}, x))$	by resolution from 2 and 3
6.	$\bot$	by resolution from 5 and 4 ■

Most resolution provers implement simplification rules to optimize proof search. Two relevant ones are

$$\frac{D \vee L \qquad C \vee (D \vee \overline{L})\sigma}{C \vee D\sigma}\ \text{subsumption resolution}$$

$$\frac{C \vee L \vee L}{C \vee L}\ \text{duplicate literal elimination}$$

The subsumption resolution rule can be seen as a combination of a resolution step with the instance $(D \vee L)\sigma$ of the first premise and zero or more duplicate literal elimination steps. The resolution rule has further side conditions that restrict the search space; these are unnecessary for soundness and are irrelevant to us. The calculus also includes a factoring rule for non-Horn clauses that plays no role in our work.

2.3 Superposition Calculus

With the resolution calculus, an equality predicate can be axiomatized, but the resulting explicit resolution inferences with the equality axioms are extremely detrimental for proof search. Robinson and Wos's paramodulation calculus [29] avoids explicit inferences with the equality axioms by integrating equality directly into the inference rules. Bachmair and Ganzinger's superposition calculus [6] improves paramodulation using ordering restrictions, selection functions, and a general redundancy concept.

In the following, equality is written in infix notation as $t \approx t'$. We write $t \not\approx t'$ for $\neg(t \approx t')$. Equality is implicitly commutative, meaning that the atoms $\mathsf{a} \approx \mathsf{b}$ and $\mathsf{b} \approx \mathsf{a}$ are considered syntactically equal.

Superposition extends resolution with two schematic inference rules. The first one is

$$\frac{C \vee s \approx s' \qquad D \vee L[u]}{(C \vee D \vee L[s'])\sigma}\ \text{superposition}$$

The notation $L[\,]$ represents a literal with a hole. The literals $L[u]$ and $L[s']$ are obtained by filling the hole in $L[\,]$ with the terms u and s', respectively. The substitution σ is an MGU of s and u. The second inference rule is

$$\frac{C \vee s \not\approx s'}{C\sigma}\ \text{equality resolution}$$

where σ is an MGU of s and s'.

The rules have further side conditions that restrict the search space. These are irrelevant to us. The calculus also includes an equality factoring rule for non-Horn clauses, which plays no role in our work.

Superposition proofs are by contradiction and usually expressed in a sequential format.

Example 4. A superposition proof from clauses 1–3 follows:

1. $x \approx \mathrm{a} \vee \mathrm{g}(\mathrm{b}, x) \approx \mathrm{f}(x)$ axiom
2. $\mathrm{c} \not\approx \mathrm{a}$ axiom
3. $\mathrm{h}(\mathrm{g}(\mathrm{b}, \mathrm{c})) \not\approx \mathrm{h}(\mathrm{f}(\mathrm{c}))$ negated goal
4. $\mathrm{c} \approx \mathrm{a} \vee \mathrm{h}(\mathrm{f}(\mathrm{c})) \not\approx \mathrm{h}(\mathrm{f}(\mathrm{c}))$ by superposition from 1 and 3
5. $\mathrm{c} \approx \mathrm{a}$ by equality resolution from 4
6. $\mathrm{a} \not\approx \mathrm{a}$ by superposition from 5 and 2
7. $\bot$ by equality resolution from 6 ■

Superposition provers use simplification rules to optimize proof search. They rely on rules such as subsumption resolution and duplicate literal elimination inherited from resolution as well as equality-specific rules such as

$$\frac{s \approx s' \qquad C \vee L[s\sigma]}{C \vee L[s'\sigma]}\ \text{demodulation}$$

where σ is an arbitrary substitution.

In practice, problems given to resolution and superposition provers are not always in clausal form and can combine arbitrary logical connectives and quantifiers in arbitrary ways. A preprocessor transforms free-form formulas to clauses. The preprocessing steps are recorded in the generated proofs, so that these match the original problems.

3 Proof Format

A proof in our format consists of a list of *axioms*, followed by zero or more *lemmas*, followed by one or more *goals*. The axioms restate the premises of the input problem, the lemmas represent intermediate facts or Horn clauses, and the goals are the statements to prove. The axioms, lemmas, and goals are numbered. Each lemma and goal is immediately followed by a subproof. Our format expresses direct proofs, in which reasoning proceeds from the axioms through the lemmas to the goals. All clauses are written using the customary Horn clause notation with implications instead of disjunctions.

The format permits both equality chains and "have/hence" blocks as subproofs. An *equality chain*, in the style of DISCOUNT and other equational provers, is a sequence $t_1 \approx t_2 \approx \cdots \approx t_n$ in which each step rewrites a subterm of the preceding term using an axiom or lemma as a left-to-right or right-to-left rewrite rule. A reflexive goal $t \approx t$ is proved by the chain t, which has no steps. A "have/hence" block represents a sequence of hyperresolution inferences. The keyword "have" introduces the first premise, and the keyword "and" introduces each additional premise. The keyword "hence" derives the conclusion from all accumulated premises by applying the specified axiom or lemma. The conclusion specified using "hence" can in turn serve as the first premise of a subsequent "hence", possibly in conjunction with "and". When the axioms alone are contradictory, a block derives $\bot$, from which every goal follows.

In the proof of a lemma of the form $A_1 \wedge \cdots \wedge A_m \rightarrow B$, where $m > 0$, a "have" or "hence" statement can be used to repeat one of the atoms A_i, or an instance therefore. In that case, the statement's proof is "by assumption". In addition, in a "hence" statement, the "rw" annotation indicates a rewrite step that uses the cited axiom or lemma as a rewrite rule on a subterm of the previous step's conclusion. By default, rewrite rules are applied from left to right; the "R→L" annotation indicates a right-to-left application.

Examples 1 and 2 are valid proofs in our format. The former is purely equational; the latter is based exclusively on hyperresolution. Example 5 combines both styles.

Example 5. The following proof establishes $\mathrm{P}(\mathrm{g}(\mathrm{a}))$ from four axioms via a lemma:

Axiom 1: $\mathrm{Q}(\mathrm{a})$

Axiom 2: $\mathrm{f}(x) \approx x$

Axiom 3: $\mathrm{g}(x) \approx \mathrm{f}(x)$

Axiom 4: $\mathrm{Q}(x) \wedge \mathrm{g}(x) \approx x \rightarrow \mathrm{P}(\mathrm{g}(\mathrm{a}))$

Lemma 5: $\mathrm{g}(x) \approx x$
Proof:
  $\mathrm{g}(x)$
$\approx$ { by axiom 3 }
  $\mathrm{f}(x)$
$\approx$ { by axiom 2 }
  x

Goal 1: $\mathrm{P}(\mathrm{g}(\mathrm{a}))$
Proof:
  have $\mathrm{Q}(\mathrm{a})$
    by axiom 1
  and $\mathrm{g}(\mathrm{a}) \approx \mathrm{a}$
    by lemma 5
  hence $\mathrm{P}(\mathrm{g}(\mathrm{a}))$
    by axiom 4 ■

Equality chains can be mechanically unfolded into sequences of "have/hence" steps if desired, but equality chains are generally more readable.

Example 6. The following proof is equivalent to the one in Example 5, with the equality chain written as "have/hence" steps instead. Our algorithm does not produce this form:

Axiom 1: $\mathrm{Q}(\mathrm{a})$

Axiom 2: $\mathrm{f}(x) \approx x$

Axiom 3: $\mathrm{g}(x) \approx \mathrm{f}(x)$

Axiom 4: $\mathrm{Q}(x) \wedge \mathrm{g}(x) \approx x \rightarrow \mathrm{P}(\mathrm{g}(\mathrm{a}))$

Goal 1: $\mathrm{P}(\mathrm{g}(\mathrm{a}))$
Proof:
  have $\mathrm{g}(\mathrm{a}) \approx \mathrm{f}(\mathrm{a})$
    by axiom 3
  hence $\mathrm{g}(\mathrm{a}) \approx \mathrm{a}$
    by rw axiom 2
  and $\mathrm{Q}(\mathrm{a})$
    by axiom 1

hence P(g(a))
by axiom 4 ■

4 Translation Algorithm

We present a translation algorithm from resolution and superposition proofs to our format as pseudocode with an accompanying description, followed by three worked examples and a correctness proof.

4.1 Description

Provers produce refutational proofs presented as DAGs of derived clauses. The translation algorithm takes such a DAG as the input proof and expands it into a proof tree T.

Following Robinson [30], we categorize each clause of the input proof as an *electron* if it consists of a single atom and as a *nucleus* otherwise. In the Horn fragment, each nucleus takes the form $A_1 \wedge \cdots \wedge A_m \rightarrow B$, where $A_1, \ldots, A_m$ are its *body atoms* and the *head* B is either an atom or $\bot$. Among electrons we distinguish between *input electrons*, which are among the input axioms, and *derived electrons*, which arise as inner nodes of the DAG. Among nuclei we similarly distinguish between *axiom nuclei*, which are among the input axioms and have a positive head, and the *goal nucleus*, which is derived from the negated goal and has head $\bot$.

Unlike a hyperresolution step, each inference of T has one or two premises. Each node of the expanded proof tree T is assigned a *position*, a binary string recording its path from the root $\bot$. The root has position ε. Appending $d \in \{0, 1\}$ to position p gives the position of the premise numbered d of the node at p, and the only premise of a unary inference is at $p1$. For each binary inference, the premise supplying a positive atom precedes the one consuming it. For resolution, the premise where the resolved atom occurs unnegated is placed at position $p0$ and the other at position $p1$. For superposition and demodulation, the premise providing the rewriting equation comes first. Between two unit equations, either of which can be read as rewriting the other, the premise listed first in the input proof is taken as the one rewritten, so the other one comes first. When neither criterion applies, the premise order of the input proof is kept, except that between two positive units (i.e., single-literal clauses) the equational one is placed first. A clause shared between multiple derivations receives a separate position for each occurrence. The resulting positions are totally ordered lexicographically. This ordering is designed to reflect the order in which literals are derived in the direct proof.

The algorithm operates under a substitution θ derived from the input proof. Since variables are local to clauses, each variable is uniquely identified by its name and the clause's position. Applying θ to every clause of T must yield a ground (i.e., variable-free) proof tree, with fresh constants for the variables that no inference binds, so θ has to agree with the unifier of every inference. For example, suppose that the prover derives the conclusion $C_0\sigma$ from premises C_1 and C_2, where σ is the unifier of the resolved literals. Then the variables of C_1 and C_2 are instantiated by σ, and the variables of $C_0\sigma$ in turn by the unifier of the inference that consumes $C_0\sigma$. The grounding substitution θ is therefore obtained by composing the unifiers of all inferences, tracing the proof from the root to the leaves.

A variable that no inference binds—for example, a variable of a unit resolved against a variable of another clause—is mapped to a fresh constant. Such a constant occurs in no axiom, so a fact derived about it holds for every value of it. When the head of a nucleus is stored as a derived electron, the fresh constants it shares with the body are therefore replaced by variables again. The head of a nucleus may moreover contain variables that do not occur in its body. These variables stay free, so that the derived electron is as general as the proof allows. Given a substitution θ, we denote by θ' the substitution obtained by replacing the fresh constants introduced by θ with variables.

The translation algorithm maintains a set *Units* containing one entry for each electron. Each entry is represented as a quadruple $(name, E, proof, pos)$, where E is the electron, and *name*, *proof*, and *pos* record its output name, stored proof, and position, respectively. In the Horn fragment, the proof tree must contain at least one positive unit leaf unless equality resolution closes a body atom; without one, there is nothing to resolve the negative body literals against, and $\bot$ can never be derived. Such clauses are precisely the input electrons, so *Units* is always nonempty from the start.

The main procedure `translate_proof` (Algorithm 1) begins with `collect_leaves`(T), which partitions the leaves into electrons and nuclei. It initializes *Units* with one entry per input electron, returns the nuclei as a list sorted by lexicographic position, and also returns a list of the goal atoms with their positions. The algorithm then processes the nuclei left to right, i.e., in order of position. Every nucleus that produced a premise of another nucleus at position p has a strictly smaller position than p; when a nucleus is reached, all electrons it depends on are therefore already in *Units*.

For each nucleus $A_1 \wedge \cdots \wedge A_m \rightarrow B$, the algorithm processes the body atoms in turn. The set of candidate electrons *Elecs* is initialized to the entries in *Units* whose position precedes the nucleus, and grows as each body atom is matched. The algorithm tries to match each body atom A_i with the electron E_i of a candidate entry U_i in *Elecs*. For each body atom A_i, the algorithm applies the substitution θ to obtain the ground atom $A_i\theta$, then calls `find_elec` (Algorithm 2) to search for a candidate electron matching $A_i\theta$. It first attempts a direct match with each candidate, establishing $A_i\theta = E_i\sigma_i$, where σ_i maps the electron's variables to terms. Since $A_i\theta$ is ground, a match instantiates only the electron. In particular, a fresh constant in $A_i\theta$ is matched by a variable of the electron. If no candidate matches directly, it retries each candidate's *rewriting chain* (`try_chain`), the sequence of rewriting steps

Algorithm 1: `translate_proof`(T)

Input: T: resolution/superposition proof tree
Output: structured proof (emitted)

$(\mathit{Units},\ \mathit{Nuclei},\ [(G_1, p_{G_1}), \ldots, (G_q, p_{G_q})]) \leftarrow$ `collect_leaves`(T)
let θ be the substitution derived from the input proof
for each $(\mathit{pos},\ \mathit{clause}) \in \mathit{Nuclei}$ **do**
  $(A_1, \ldots, A_m \to B) \leftarrow \mathit{clause}$
  if $B \neq \bot$ **then**
    $\mathit{Elecs} \leftarrow \{\, U \in \mathit{Units} \mid U.\mathit{pos} \prec \mathit{pos} \,\}$
    $\mathit{block} \leftarrow$ ""
    for $i = 1$ ***to*** m **do**
      $(U_i,\ \sigma_i) \leftarrow$ `find_elec`$(A_i\theta,\ \mathit{Elecs},\ \mathit{Units},\ \mathit{pos})$
      $E_i \leftarrow U_i.E$
      $\mathit{Elecs} \leftarrow \mathit{Elecs} \cup \{\, U_i[E \mapsto E_i\sigma_i] \,\}$
      $\mathit{block}_i \leftarrow$ `make_block`$(E_i,\ \sigma_i,\ [\,],\ \mathit{Units})$
      if $i = 1$ **then**
        $\mathit{block} \leftarrow \mathit{block}_i$
      else
        if $A_i\theta \notin \mathit{Units}$ **then**
          $\mathit{Units} \leftarrow \mathit{Units} \cup \{(\mathit{nil},\ A_i\theta,\ \mathit{block}_i,\ \mathit{nil})\}$
        $\mathit{name}_i \leftarrow$ `name_of`$(A_i\theta,\ \mathit{Units})$
        $\mathit{block} \leftarrow \mathit{block} \cdot$ "and $A_i\theta$ by name_i"
  if $B = \bot$ **then**
    for $j = 1$ ***to*** q **do**
      if $\exists\ (\mathit{name},\ E,\ \mathit{proof},\ \mathit{pos}) \in \mathit{Units}, \rho_0, \mathit{rw}$ *s.t.* $E\rho_0 \to^*_{\mathit{rw}} G_j\theta$, *with rw empty or the rewriting chain of* E **then**
        $\mathit{block} \leftarrow$ `make_block`$(E,\ \rho_0,\ \mathit{rw},\ \mathit{Units})$
        emit "Goal j: $G_j\theta$"
        emit "Proof:" $\cdot$ *block*
      else
        $(l \approx r) \coloneqq G_j\theta$, read as $G_j\theta \approx$ true if G_j is no equation
        $\mathit{chain} \leftarrow$ `get_chain`$(l,\ r,\ p_{G_j},\ \mathit{Units})$
        emit "Goal j: $G_j\theta$"
        emit "Proof: l"
        for each $(\mathit{name},\ \mathit{dir},\ t) \in \mathit{chain}$ **do**
          emit "$\approx t$ by *name*" $\cdot$ ("R→L" if $\mathit{dir} = R{\to}L$)
    return
  else
    $\mathit{block} \leftarrow \mathit{block} \cdot$ "hence $B\theta'$ by `name`(clause)"
    $\mathit{Units} \leftarrow \mathit{Units} \cup \{(\mathit{nil},\ B\theta',\ \mathit{block},\ \mathit{pos})\}$

recorded when that electron was derived or simplified in the input proof. If the electron rewritten along this chain matches $A_i\theta$, it is added to *Units* as a new unit, proved by the chain. The input proof may also rewrite the body atom itself before resolving it, and then $A_i\theta$ is rewritten along these recorded steps and matched with an electron, and the steps are appended in reverse, so the block still ends with $A_i\theta$. One of the two attempts succeeds, since under θ the electron resolved against A_i in T matches $A_i\theta$, possibly after its recorded rewrites. After each match, the instantiated

Algorithm 2: `find_elec`$(A_i\theta,\ \mathit{Elecs},\ \mathit{Units},\ \mathit{pos})$

Input: $A_i\theta$: ground body atom; *Elecs*: candidate electrons; *Units*: unit set; *pos*: clause position
Output: (U_i, σ_i)

for each $U \in \mathit{Elecs}$ **do**
  if ***let*** $\sigma_i \leftarrow$ `try_match`$(A_i\theta,\ U.E)$ **then**
    return $(U,\ \sigma_i)$
for each $U \in \mathit{Elecs}$ **do**
  if ***let*** $(U', \sigma_i) \leftarrow$ `try_chain`$(A_i\theta,\ U)$ **then**
    $\mathit{Units} \leftarrow \mathit{Units} \cup \{U'\}$
    return $(U',\ \sigma_i)$

Algorithm 3: `make_block`$(E,\ \rho,\ \mathit{rw},\ \mathit{Units})$

Input: E: atom; ρ: substitution; *rw*: list of rewrite steps; *Units*: unit set
Output: subproof establishing $E\rho$ via rewriting

if $(\mathit{nil},\ E,\ \mathit{proof},\ \mathit{pos}) \in \mathit{Units}$ ***and*** $\neg$ `is_eq_chain`(proof) **then**
  $\mathit{block} \leftarrow \mathit{proof}\ \rho$
else
  $\mathit{block} \leftarrow$ "have $E\rho$ by `name_of`$(E,\ \mathit{Units})$"
for each $(\mathit{name},\ \mathit{dir},\ t) \in \mathit{rw}$ **do**
  $\mathit{block} \leftarrow \mathit{block} \cdot$ "hence t by rw *name*" $\cdot$ ("R→L" if $\mathit{dir} = R{\to}L$)
return *block*

electron is added to *Elecs*, making it available as a candidate for subsequent body atoms within the same nucleus.

Next, for each body atom, `make_block` (Algorithm 3) constructs the corresponding subproof, which is either the instantiation by σ_i of a stored proof or a single "have $E_i\sigma_i$ by *name*" line referencing the matching axiom or lemma. As a readability heuristic, equality chains are always promoted to lemmas and referenced by name instead of inlined. If rewrite steps are present, each is appended as a "hence t by rw *name*" line, where *name* is the name of the axiom or lemma applied in the rewrite step, with an explicit direction if the application is right to left. For the first body atom $A_1\theta$, the subproof is inlined as the start of the nucleus proof. For each subsequent $A_i\theta$, a lemma name is assigned to $A_i\theta$, and the subproof becomes the proof of that lemma, which the "and" premise cites. A subproof that only cites an axiom or lemma gives that name instead. A final "hence $B\theta'$" closes the nucleus proof. The assembled subproof encodes one hyperresolution step. The set *Units* is then extended with a new derived electron for $B\theta'$, with the fresh constants introduced by θ replaced by variables, so the electron may be nonground. The new entry carries no lemma name initially and records the nucleus's position. A name is assigned when the electron is cited as an "and" premise or carries an equality chain proof.

Among the nuclei processed in the main loop, the negated goal nucleus has the form $G_1 \wedge \cdots \wedge G_q \to \bot$, i.e., $B = \bot$. When it is reached, for each G_j the algorithm searches *Units*

for an electron matching $G_j\theta$ under some substitution ρ_0, assembles the proof with `make_block`, and emits it as the goal's proof. The electron may first be rewritten along its rewriting chain *rw*; `make_block` appends the rewrite steps. For goals $l \approx r$ where no such electron is found, with an atom $G_j\theta$ read as $G_j\theta \approx \mathsf{true}$, `get_chain` returns a list of rewriting steps establishing $l \approx r$. Each step is a tuple $(\mathit{name}, \mathit{dir}, t)$ recording the applied unit, direction, and intermediate term. For an atom, the chain ends with a fact, and the proof is printed as that fact followed by the rewrite steps in reverse.

The algorithm yields unit lemmas, by naming equality chains and derived electrons cited as premises. The subproof above each shared clause is duplicated at every use, leading in the worst case to an exponential explosion in proof size. To avoid this, we extend the algorithm to perform *lemma introduction*. Whenever the input DAG contains a derived Horn clause C with a head that is used at least twice in the DAG, the algorithm makes it a lemma. It negates the clause, yielding the formula $\exists x_1, \ldots, x_k.\, \neg C$, where the variables x_i correspond to C's free variables, and replaces each x_i by a fresh Skolem constant c_i. For $C = A_1 \wedge \cdots \wedge A_m \to B$, with $m = 0$ for a unit, this yields the ground atoms $A_1\sigma, \ldots, A_m\sigma$ and the negated ground atom $\neg B\sigma$, where σ maps each x_i to c_i. The algorithm then extracts the derivation of C from the input proof as a refutation of these clauses from the problem's axioms and translates it recursively, citing each $A_i\sigma$ by assumption. The constants c_i are treated as arbitrary but fixed constants throughout the proof steps. The statement of the resulting lemma is C. The resulting proof, with each c_i replaced back by x_i, is emitted as a lemma, and C is thereafter treated as an axiom when translating the input proof.

Example 7. We start with the following resolution proof:

1. $\mathsf{U}(\mathsf{b})$	axiom
2. $\neg\mathsf{U}(x) \vee \mathsf{Q}(x)$	axiom
3. $\mathsf{S}(\mathsf{a})$	axiom
4. $\neg\mathsf{S}(x) \vee \mathsf{P}(x)$	axiom
5. $\neg\mathsf{P}(x) \vee \neg\mathsf{Q}(y) \vee \mathsf{R}(x, y)$	axiom
6. $\neg\mathsf{R}(\mathsf{a}, \mathsf{b})$	negated goal
7. $\mathsf{P}(\mathsf{a})$	by resolution from 4 and 3
8. $\mathsf{Q}(\mathsf{b})$	by resolution from 2 and 1
9. $\neg\mathsf{P}(\mathsf{a}) \vee \neg\mathsf{Q}(\mathsf{b})$	by resolution from 5 and 6
10. $\neg\mathsf{Q}(\mathsf{b})$	by resolution from 9 and 7
11. $\bot$	by resolution from 10 and 8

The above proof is essentially a DAG. The corresponding proof tree is

$\bot$	(pos. ε)
$\quad\mathsf{Q}(\mathsf{b})$	(pos. `0`)
$\quad\quad\mathsf{U}(\mathsf{b})$	(axiom 1, pos. `00`)
$\quad\quad\neg\mathsf{U}(x) \vee \mathsf{Q}(x)$	(axiom 2, pos. `01`)
$\quad\neg\mathsf{Q}(\mathsf{b})$	(pos. 1)
$\quad\quad\mathsf{P}(\mathsf{a})$	(pos. `10`)
$\quad\quad\quad\mathsf{S}(\mathsf{a})$	(axiom 3, pos. `100`)
$\quad\quad\quad\neg\mathsf{S}(x) \vee \mathsf{P}(x)$	(axiom 4, pos. `101`)
$\quad\quad\neg\mathsf{P}(\mathsf{a}) \vee \neg\mathsf{Q}(\mathsf{b})$	(pos. 11)
$\quad\quad\neg\mathsf{P}(x) \vee \neg\mathsf{Q}(y) \vee \mathsf{R}(x, y)$	(axiom 5, pos. `110`)
$\quad\neg\mathsf{R}(\mathsf{a}, \mathsf{b})$	(goal, pos. 111)

Axioms 1 and 3 are input electrons. Axioms 2, 4, and 5 and the negated goal are nuclei, with goal atom $\mathsf{R}(\mathsf{a}, \mathsf{b})$. *Units* is initialized with

$$\{(\text{“axiom 1”}, \mathsf{U}(\mathsf{b}), \mathit{nil}, \mathtt{00}),\ (\text{“axiom 3”}, \mathsf{S}(\mathsf{a}), \mathit{nil}, \mathtt{100})\}$$

Axioms 2, 4, and 5 each use a variable x. This does not mean that they refer to the same variable; the prover names variables independently in each clause of the input proof. We distinguish them by subscripting with the clause's position. The substitution θ derived from the input proof is then

$$\theta = \{x_{\mathtt{01}} \mapsto \mathsf{b},\ x_{\mathtt{101}} \mapsto \mathsf{a},\ x_{\mathtt{110}} \mapsto \mathsf{a},\ y_{\mathtt{110}} \mapsto \mathsf{b}\}$$

Algorithm 1 processes the nuclei from left to right in the proof tree. The first nucleus is $\neg\mathsf{U}(x) \vee \mathsf{Q}(x)$, at position `01`. For $\mathsf{U}(x)$, applying θ gives $\mathsf{U}(\mathsf{b})$. The algorithm searches *Units* for a matching electron at an earlier position and finds a direct match with axiom 1 at position `00`. It then constructs a subproof of the head instance $\mathsf{Q}(\mathsf{b})$, opening with “have $\mathsf{U}(\mathsf{b})$ by axiom 1” and closing with “hence $\mathsf{Q}(\mathsf{b})$ by axiom 2”. The derived electron $\mathsf{Q}(\mathsf{b})$ is added to *Units*.

Next, the algorithm processes $\neg\mathsf{S}(x) \vee \mathsf{P}(x)$ at position `101`. For the body atom $\mathsf{S}(x)$, applying θ gives $\mathsf{S}(\mathsf{a})$, which matches axiom 3 in *Units*. A subproof of $\mathsf{P}(\mathsf{a})$ is constructed in the same way as before, and the derived electron $\mathsf{P}(\mathsf{a})$ is added to *Units*.

The algorithm then processes $\neg\mathsf{P}(x) \vee \neg\mathsf{Q}(y) \vee \mathsf{R}(x, y)$ at position `110`. After it applies θ, the two body atoms become $\mathsf{P}(\mathsf{a})$ and $\mathsf{Q}(\mathsf{b})$. $\mathsf{P}(\mathsf{a})$ matches the derived electron $\mathsf{P}(\mathsf{a})$. Its stored subproof is inlined. $\mathsf{Q}(\mathsf{b})$ matches the derived electron $\mathsf{Q}(\mathsf{b})$, but since it is the second body atom, it is promoted to a new lemma, lemma 6. Its proof is the subproof stored for $\mathsf{Q}(\mathsf{b})$. The resulting proof opens with “have $\mathsf{S}(\mathsf{a})$ by axiom 3”, continues with “hence $\mathsf{P}(\mathsf{a})$ by axiom 4”, adds “and $\mathsf{Q}(\mathsf{b})$ by lemma 6”, and closes with “hence $\mathsf{R}(\mathsf{a}, \mathsf{b})$ by axiom 5”. The derived electron $\mathsf{R}(\mathsf{a}, \mathsf{b})$ is added to *Units*.

Finally, the algorithm processes the negated goal $\neg\mathsf{R}(\mathsf{a}, \mathsf{b})$ at position 111. This clause has the form $\mathsf{R}(\mathsf{a}, \mathsf{b}) \to \bot$, so its body atom $\mathsf{R}(\mathsf{a}, \mathsf{b})$ is the goal atom G_1. Since $B = \bot$ for this nucleus, the algorithm matches $\mathsf{R}(\mathsf{a}, \mathsf{b})$ with the stored electron $\mathsf{R}(\mathsf{a}, \mathsf{b})$. Its subproof is emitted directly as goal 1. The result is the proof presented in Example 2. ■

Example 8. We start with the following superposition proof:

1. $\mathsf{g}(x) \approx \mathsf{f}(x)$	axiom
2. $\mathsf{f}(x) \approx x$	axiom
3. $\mathsf{g}(\mathsf{g}(\mathsf{a})) \not\approx \mathsf{a}$	negated goal
4. $\mathsf{g}(x) \approx x$	by superposition from 2 and 1
5. $\mathsf{g}(\mathsf{a}) \not\approx \mathsf{a}$	by superposition from 3 and 4
6. $\mathsf{a} \not\approx \mathsf{a}$	by superposition from 5 and 4
7. $\bot$	by equality resolution from 6

Before expanding this proof into a tree, the algorithm performs lemma introduction. Clause 4, $g(x) \approx x$, is not an axiom and is used twice in the proof, as a premise of clauses 5 and 6. The algorithm negates the clause and replaces x by a fresh Skolem constant c, yielding $g(c) \not\approx c$. Applying the translation recursively yields the chain $g(c) \approx f(c) \approx c$ from axioms 1 and 2. Replacing c back by x gives the chain $g(x) \approx f(x) \approx x$. The algorithm promotes this chain to a named lemma, lemma 3, which states $g(x) \approx x$ and has the chain as its proof, and replaces clause 4 by lemma 3 before expanding the proof into a tree. As a result, the derivation of clause 4 is translated only once.

The corresponding proof tree is

$\bot$ (pos. ε)
- $a \not\approx a$ (pos. 1)
  - $g(x) \approx x$ (lemma 3, pos. 10)
  - $g(a) \not\approx a$ (pos. 11)
    - $g(x) \approx x$ (lemma 3, pos. 110)
    - $g(g(a)) \not\approx a$ (goal, pos. 111)

Axioms 1 and 2 occur only in the proof of lemma 3, which acts as an input electron. The negated goal is $g(g(a)) \not\approx a$. *Units* is initialized with lemma 3 at its first position

$$\{(\text{“lemma 3”}, g(x) \approx x, g(x) \approx f(x) \approx x, 10)\}$$

The only nucleus is the negated goal at position 111, with body atom $g(g(a)) \approx a$ and head $\bot$. Since this nucleus is ground, θ leaves it unchanged. No entry in *Units* can match this atom directly. The algorithm finds the equality chain $g(g(a)) \approx g(a) \approx a$ using lemma 3 twice, with $x \mapsto g(a)$ and $x \mapsto a$ respectively, and emits it as the goal proof. The result is the proof presented in Example 1. ■

Example 9. We start with the following superposition proof:

1.	$Q(a)$	axiom
2.	$f(x) \approx x$	axiom
3.	$g(x) \approx f(x)$	axiom
4.	$\neg Q(x) \vee g(x) \not\approx x \vee P(g(a))$	axiom
5.	$\neg P(g(a))$	negated goal
6.	$g(x) \approx x$	by superposition from 3 and 2
7.	$\neg Q(x) \vee g(x) \not\approx x$	by resolution from 4 and 5
8.	$\neg Q(x)$	by resolution from 7 and 6
9.	$\bot$	by resolution from 1 and 8

The corresponding proof tree is

$\bot$ (pos. ε)
- $Q(a)$ (axiom 1, pos. 0)
- $\neg Q(x)$ (pos. 1)
  - $g(x) \approx x$ (pos. 10)
    - $f(x) \approx x$ (axiom 2, pos. 100)
    - $g(x) \approx f(x)$ (axiom 3, pos. 101)
  - $\neg Q(x) \vee g(x) \not\approx x$ (pos. 11)
    - $\neg Q(x) \vee g(x) \not\approx x \vee P(g(a))$ (axiom 4, pos. 110)
    - $\neg P(g(a))$ (goal, pos. 111)

Axioms 1, 2, and 3 are input electrons, whereas axiom 4 and the negated goal are nuclei, with goal atom $P(g(a))$. *Units* is initialized with

$$\{(\text{“axiom 1”}, Q(a), \mathit{nil}, 0),\ (\text{“axiom 2”}, f(x) \approx x, \mathit{nil}, 100),\ (\text{“axiom 3”}, g(x) \approx f(x), \mathit{nil}, 101)\}$$

On the leaves, the substitution θ is

$$\theta = \{x_{100} \mapsto a,\ x_{101} \mapsto a,\ x_{110} \mapsto a\}$$

Algorithm 1 processes the nucleus $\neg Q(x) \vee g(x) \not\approx x \vee P(g(a))$ at position 110, i.e., $Q(x) \wedge g(x) \approx x \rightarrow P(g(a))$. Under θ, its body atoms become $Q(a)$ and $g(a) \approx a$. The first matches axiom 1 directly. The second matches neither equational electron directly. The input proof derives $g(x) \approx x$ by rewriting axiom 3 using axiom 2, which replaces $f(x)$ by x. The algorithm reads this step as the chain $g(x) \approx f(x) \approx x$. This chain's conclusion $g(x) \approx x$ matches the body atom $g(a) \approx a$ under the substitution $x \mapsto a$. The algorithm promotes the chain to a named lemma, lemma 5, which states $g(x) \approx x$ and has the chain as its proof. It then assembles a subproof for $P(g(a))$, opening with “have $Q(a)$ by axiom 1”, continuing with “and $g(a) \approx a$ by lemma 5” and closing with “hence $P(g(a))$ by axiom 4”, and adds $P(g(a))$ to *Units* as a derived electron.

Finally, the negated goal $\neg P(g(a))$ at position 111 is processed. Its body atom $P(g(a))$ matches the derived electron in *Units*, and the corresponding proof is emitted as the goal proof. The result is the proof presented in Example 5. ■

4.2 Correctness

We have to prove that, given a resolution or superposition proof as input, the algorithm always produces a correct proof in our format as output.

Let us start with some theoretical considerations. Let T be an input proof tree for a Horn problem using any of the rules (subsumption) resolution, duplicate literal elimination, superposition, demodulation, and equality resolution. To transform T, we must keep track of the ancestry of literals occurring in the tree. To this end, we assign sets of label sets to literal occurrences in T. For sets of sets $\mathcal{A}$ and $\mathcal{B}$, define $\mathcal{A} \sqcup \mathcal{B} = \{ x \cup y \mid x \in \mathcal{A},\ y \in \mathcal{B} \}$. We now proceed in the following way: If L is a literal in a leaf clause in T, we assign the set $\{\{\ell\}\}$ to it, where ℓ is an arbitrary unique identifier. (If the same clause occurs at several leaf positions in T, or if one literal occurs multiple times in a clause, the different occurrences of literals get different identifiers.) If a literal L in the conclusion of a superposition or demodulation inference is obtained by combining two literals with sets of label sets S_1 and S_2, we assign the set $S_1 \sqcup S_2$ to L. If a literal L in the conclusion of a duplicate literal elimination is obtained by combining two literals with sets of label sets S_1 and S_2,

we assign the set $S_1 \cup S_2$ to L. (Note that in the Horn case, a duplicated literal is necessarily negative.) Otherwise, that is, if a literal L in the conclusion of an inference results from a literal with a set of label sets S in a premise by mere copying and possibly instantiation, we assign the set S to L. Subsumption resolution inferences are treated like combinations of resolution and duplicate literal elimination inferences.

The function *leaflit* maps a label ℓ to the unique literal in a leaf clause to which $\{\{\ell\}\}$ is assigned. The function *leafpos* maps a label ℓ to the unique position p in T such that a literal L in a leaf clause at position p is assigned the set $\{\{\ell\}\}$. We call a label ℓ positive or negative, respectively, if $\mathit{leaflit}(\ell)$ is a positive or negative literal. We define an ordering $\succ$ on labels in such a way that $\ell \succ \ell'$ if $\mathit{leafpos}(\ell)$ is lexicographically larger than $\mathit{leafpos}(\ell')$.

Lemmas 1 and 2 can be proved by induction over the proof structure and case analysis over the possible inferences:

Lemma 1. *If L is a positive literal in a clause in T, then it is assigned a singleton set $\{M\}$ and every label in M is positive.*

Lemma 2. *If L is a negative literal in a clause in T, then it is assigned a set S such that every $M \in S$ contains exactly one negative label ℓ and $\ell \succ \ell'$ for all positive labels ℓ' in M.*

We say that a negative label ℓ *disappears* in T using a set of positive labels M if there is an equality resolution inference

$$\frac{C \vee s \not\approx s'}{C\sigma}$$

in T where $s \not\approx s'$ is assigned a set $\{\{\ell\} \cup M\} \cup S$, or if there is a resolution inference

$$\frac{C_1 \vee A_1 \qquad C_2 \vee \neg A_2}{C_1\sigma \vee C_2\sigma}$$

in T where A_1 is assigned a set $\{M_1\}$, $\neg A_2$ is assigned a set $\{\{\ell\} \cup M_2\} \cup S$, and $M = M_1 \cup M_2$.

Since the input proof tree has the empty clause at the root, every negative label must eventually disappear. Using Lemma 2, we obtain the following lemma:

Lemma 3. *If a negative literal in a leaf clause in T is assigned a set $\{\{\ell\}\}$, then ℓ disappears in T in an equality resolution inference or resolution inference using a finite set of positive labels M and $\ell \succ \ell'$ for all labels ℓ' in M.*

If T is a ground input proof tree, then every literal occurring anywhere in T is a logical consequence of its ancestor literals. The following lemma formalizes this fact. It is again easily proved by induction over the proof structure and case analysis over the possible inferences:

Lemma 4. *Assume that T is a ground resolution or superposition proof tree. If a literal L_0 in a clause in T is assigned a set $\{\{\ell_1, \ldots, \ell_n\}\} \cup S$, then $\{\mathit{leaflit}(\ell_1), \ldots, \mathit{leaflit}(\ell_n)\} \models L_0$.*

Analogously, if a literal $s \not\approx s$ is eliminated in a ground equality resolution inference, then the ancestor literals of $s \not\approx s$ are inconsistent, and if literals A and $\neg A$ are eliminated in a resolution inference, then the ancestor literals of A and $\neg A$ are inconsistent:

Lemma 5. *Assume that T is a ground resolution or superposition proof tree. If a negative label ℓ_0 disappears in T using a finite set of positive labels $\{\ell_1, \ldots, \ell_n\}$, then $\{\mathit{leaflit}(\ell_0), \ldots, \mathit{leaflit}(\ell_n)\} \models \bot$.*

Theorem 1. *Assume that T is a resolution or superposition proof tree and that $T\theta$ is a ground resolution or superposition proof tree obtained from T by applying some grounding substitution θ to all clauses in T. Let $(C_k)_{1 \le k \le n}$ be the sequence of all leaf clauses in T that are not positive unit clauses, in ascending order with respect to the position order. For $1 \le k \le n$ let $C_k = D_k \vee B_k$, where $D_k = \neg A_{k1} \vee \cdots \vee \neg A_{km_k}$ is a disjunction of negative literals and B_k is either a positive literal or $\bot$. Let the substitution θ'_k be defined like θ, except that θ'_k maps all variables that occur in B_k but not in D_k to themselves. Let p_k be the position of C_k in T and let E_k be the set of all positive unit leaf clauses that occur in T at positions smaller than p_k. For $1 \le k \le n$ and $0 \le i \le m_k$ let*

$$U_{ki} = E_k \cup \bigcup_{\substack{1 \le j < k,\\ 1 \le l \le m_j}} \{A_{jl}\theta\} \cup \bigcup_{1 \le l < i} \{A_{kl}\theta\} \cup \bigcup_{1 \le j < k} \{B_j\theta'_j\}.$$

Then the following properties hold:

(i) For every $1 \le k \le n$ and $1 \le i \le m_k$, there exists a rewrite proof of $A_{ki}\theta$ using equations from U_{ki}.

(ii) For every $1 \le k \le n$, $\bigcup_{1 \le l \le m_k} \{A_{kl}\theta\} \cup \{C_k\} \models B_k\theta'_k$.

Proof. (i) The literal $\neg A_{ki}\theta$ is a negative literal occurring in a leaf clause in $T\theta$ at position p_k. By Lemma 3, its label ℓ must disappear in $T\theta$ using positive labels that are all smaller than ℓ. By Lemma 5, this implies that there are positive literals $A'_1\theta, \ldots, A'_h\theta$ occurring in $T\theta$ at leaf positions smaller than p_k such that $\{\neg A_{ki}\theta, A'_1\theta, \ldots, A'_h\theta\} \models \bot$ or, in other words, $\{A'_1\theta, \ldots, A'_h\theta\} \models A_{ki}\theta$. Now for each A'_l there are two possibilities: If A'_l is a positive unit leaf clause in T, then we have $A'_l \in E_k \subseteq U_{ki}$ and the unit clause A'_l entails its ground instance $A'_l\theta$, therefore $U_{ki} \models A'_l\theta$. Otherwise, A'_l is a positive literal in a nonunit leaf clause in T, that is, a positive literal B_j in a leaf clause C_j with $j < k$. Then $A'_l\theta = B_j\theta$ is a ground instance of the (nonleaf) unit clause $B_j\theta'_j$ and therefore entailed by it. Since $B_j\theta'_j \in U_{ki}$, we again get $U_{ki} \models A'_l\theta$. In a Horn refutation tree, an inference with a negative premise has a negative conclusion, and every inference with a negative conclusion has exactly one negative premise, which comes second, since the premise supplying a positive literal precedes the one consuming it. Hence C_n is the only leaf clause with head $\bot$, and U_{ki} contains only positive unit clauses. Reading each nonequational atom A as the equation $A \approx \mathrm{true}$, U_{ki} is a set of unit equations that

entails $A_{ki}\theta$, so there is a rewrite proof of $A_{ki}\theta$ using U_{ki} by Birkhoff's theorem.

(ii) This follows immediately by hyperresolution. □

Putting together the rewrite proofs obtained for part (i) and the hyperresolution steps obtained for part (ii), we obtain a proof with the desired shape. Note that the last leaf clause that is not a positive unit, that is, C_n, must necessarily have the head $\bot$, so that the "if $B = \bot$" part is ultimately reached. When C_n is the negated goal, its body atoms are the goal atoms, so each goal $G_j\theta$, where $1 \le j \le m_n$, is proved by the rewrite proof that part (i) gives for it.

In the algorithm, the set *Nuclei* contains $\{ C_k \mid 1 \le k \le n \}$. *Units* is initially the set of positive unit leaf clauses in T and is extended as each nucleus is processed. When the body atom A_{ki} is processed, the set *Elecs* contains E_k and every $B_j\theta'_j$ with $j < k$, which is all that part (i) uses.

The rewrite proofs are obtained from the equations in *Units*. Part (i) ensures that when C_k is processed, a rewrite proof of each body atom $A_{ki}\theta$ always exists. The algorithm finds it by matching $A_{ki}\theta$ directly with an electron in *Units* or by following *rewriting chains* extracted from the input proof. Note that the proof of Theorem 1 would also work if $\bigcup_{1\le j<k,\,1\le l\le m_j}\{A_{jl}\theta\} \cup \bigcup_{1\le l<i}\{A_{kl}\theta\}$ were omitted from the definition of U_{ki}. In the implementation, however, it is convenient to include these atoms because they sometimes lead to shorter proofs.

5 Implementation

Our approach is implemented in a tool called Tälja, developed in Haskell. The overall process is as follows: The E superposition prover, the Twee equational prover, or the Vampire superposition prover is invoked on a Horn problem. The output is a resolution or superposition proof in the CNF, FOF, or TFF subformat of TSTP (Thousands of Solutions from Theorem Provers) [38]. Tälja takes this output as its input and generates a proof in our proposed format. It can also output the proof as a TSTP derivation or as a Lean file. For a FOF or TFF conjecture of the form $A_1 \wedge \cdots \wedge A_m \to B$, the hypotheses $A_1, \ldots, A_m$ are listed with the axioms and B is the goal.

TSTP proofs generally do not record unifiers, so Tälja reconstructs θ by solving all inferences of the proof tree at once. Each inference is read in every way the prover may have performed it, that is, which literals were resolved, which subterm was rewritten, and in which orientation an equation was used. This includes an equality resolution the prover folds into the step, a rewrite at a position that is a variable in the premise, and two body atoms it merges before resolving. In a Twee step that rewrites atoms with an equation between atoms, each atom P is read as the equation $P \approx \mathsf{true}$. An inference that no reading explains is left out, and variables that no inference binds become fresh constants.

The implementation follows Algorithm 1, with a few refinements. For each body atom, Tälja first tries the instances matched for earlier body atoms of the same nucleus, then the electron that the input proof itself used at this inference, then derived electrons, and finally axioms and lemmas. Moreover, matching is performed with backtracking. A body atom may match several electrons, and the step assembled from an early choice for A_i may be rejected by the validation described below. In that case, the algorithm retries with a different electron for A_i. In addition, each assembled hyperresolution step is validated before it is emitted. The applied rule, instantiated so that its body atoms coincide with the matched premises, must derive the emitted conclusion. When this check fails, the nucleus is skipped and its head is not stored. The translation fails only when a goal is then left without a proof. This does not contradict the correctness theorem, which assumes that θ is given, since the implementation has to reconstruct θ from the input proof. Finally, unreferenced lemmas are pruned.

The implementations of `try_chain` (Algorithm 2) and `get_chain` (Algorithm 1, line 28) both read the rewriting information present in the input proof. Resolution and superposition provers often simplify clauses by applying unit equalities before an inference step. These steps appear in the TSTP proof as forward or backward demodulation, rewriting, definition unfolding, or superposition inferences from a unit equation. For each leaf in the proof tree, Tälja extracts the sequence of such steps along the path to that leaf, yielding a *rewriting chain*. Each step is read from the conclusion of its inference in the input proof, which determines the applied equality, its instance, the direction, and the rewritten subterm. In `find_elec`, when no direct match succeeds, the candidate electron is rewritten along its chain and the match is retried. In `get_chain`, for equational goals $l \approx r$, the SplitChain construction of Denzinger and Schulz [15] splits the rewriting chain apart at the goal's proof-tree position into steps reducing the left-hand side l and steps reducing the right-hand side r, meeting at a common intermediate term t and yielding $l \to^* t \leftarrow^* r$.

6 Integration with Twee

Twee natively supports only equational problems, but it can also process Horn problems via an encoding [11], which we summarize here. Given a Horn problem, Twee first transforms it to eliminate predicate literals and clauses with more than one negative literal. Specifically, it transforms the predicate literal $\mathsf{P}(t_1, \ldots, t_n)$ into the equation $\mathsf{P}(t_1, \ldots, t_n) \approx \mathsf{true}$, where true is a fresh constant, and transforms the clause $t_1 \approx u_1 \wedge \ldots \wedge t_n \approx u_n \to v \approx w$ into $\mathsf{tuple}_n(t_1, \ldots, t_n) \approx \mathsf{tuple}_n(u_1, \ldots, u_n) \to v \approx w$, where tuple_n is a fresh function symbol.

We are left with a Horn problem where all clauses either are unit equations or have the form $t \approx u \to v \approx w$. To

encode the remaining clauses, Twee introduces the function $\mathsf{ifeq}(x, y, z, w)$, which in the intended model expresses "if $x \approx y$ then z else w", and the following axiom:

$$\mathsf{ifeq}(x, x, y, z) \approx y \qquad \text{(IFEQ-ELIM)}$$

It then encodes $t \approx u \to v \approx w$ as the equation

$$\mathsf{ifeq}(t, u, v, w) \approx w \qquad \text{(IFEQ-INTRO)}$$

Notice that if we can prove $t \approx u$, then we can deduce $v \approx w$ as follows: $v \approx \mathsf{ifeq}(t, t, v, w) \approx \mathsf{ifeq}(t, u, v, w) \approx w$.

Example 10. Twee encodes the problem presented in Example 2 to the following equations and finds a proof similar to the following one. (Twee's proof is slightly different in that it does not introduce lemma 6.)

Axiom 0: $\mathsf{ifeq}(x, x, y, z) \approx y$

Axiom 1: $\mathsf{U}(\mathsf{b}) \approx \mathsf{true}$

Axiom 2: $\mathsf{ifeq}(\mathsf{U}(x), \mathsf{true}, \mathsf{Q}(x), \mathsf{true}) \approx \mathsf{true}$

Axiom 3: $\mathsf{S}(\mathsf{a}) \approx \mathsf{true}$

Axiom 4: $\mathsf{ifeq}(\mathsf{S}(x), \mathsf{true}, \mathsf{P}(x), \mathsf{true}) \approx \mathsf{true}$

Axiom 5: $\mathsf{ifeq}(\mathsf{tuple}(\mathsf{P}(x), \mathsf{Q}(y)), \mathsf{tuple}(\mathsf{true}, \mathsf{true}),$
$\mathsf{R}(x, y), \mathsf{true}) \approx \mathsf{true}$

Lemma 6: $\mathsf{Q}(\mathsf{b}) \approx \mathsf{true}$
Proof:
$\mathsf{Q}(\mathsf{b})$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}(\mathsf{true}, \mathsf{true}, \mathsf{Q}(\mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 1 }
$\mathsf{ifeq}(\mathsf{U}(\mathsf{b}), \mathsf{true}, \mathsf{Q}(\mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 2 }
true

Goal 1: $\mathsf{R}(\mathsf{a}, \mathsf{b}) \approx \mathsf{true}$
Proof:
$\mathsf{R}(\mathsf{a}, \mathsf{b})$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}(\mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(\mathsf{a}, \mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 4 }
$\mathsf{ifeq}(\mathsf{tuple}(\mathsf{ifeq}(\mathsf{S}(\mathsf{a}), \mathsf{true}, \mathsf{P}(\mathsf{a}), \mathsf{true}), \mathsf{true}),$
$\mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(\mathsf{a}, \mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 3 }
$\mathsf{ifeq}(\mathsf{tuple}(\mathsf{ifeq}(\mathsf{true}, \mathsf{true}, \mathsf{P}(\mathsf{a}), \mathsf{true}), \mathsf{true}),$
$\mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(\mathsf{a}, \mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}(\mathsf{tuple}(\mathsf{P}(\mathsf{a}), \mathsf{true}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(\mathsf{a}, \mathsf{b}), \mathsf{true})$
$\approx$ { by lemma 6 }
$\mathsf{ifeq}(\mathsf{tuple}(\mathsf{P}(\mathsf{a}), \mathsf{Q}(\mathsf{b})), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(\mathsf{a}, \mathsf{b}), \mathsf{true})$
$\approx$ { by axiom 5 }
true ■

This proof is not entirely satisfactory since it reasons about the encoded equational problem rather than the original Horn problem. We would much prefer a proof similar to Example 2. Therefore, we have implemented a proof transformation that *decodes* a proof of an encoded problem back into a proof of the original problem. The decoded proof is based entirely on rewriting and resolution, so we can then use the algorithm of Section 4 to turn it into a human-readable proof.

We discuss completeness of our decoding algorithm in Section 6.3; we believe that it is incomplete but that it can be refined to be complete. In practice, it fails only rarely.

6.1 A Naive Approach

To motivate the decoding algorithm, we first show why a naive approach fails. Suppose that we define a function D that decodes an equation involving ifeq to its meaning as a Horn clause. We might then define a function that decodes a proof of $t \approx u$ to a proof of $D(t \approx u)$ by recursion on the proof structure. For example, suppose that the encoded proof ends in a transitivity step

$$\frac{t \approx u \qquad u \approx v}{t \approx v}$$

By recursion, we get proofs of $D(t \approx u)$ and $D(u \approx v)$, and we need to construct a proof of $D(t \approx v)$. That is, the following inference rule must be admissible:

$$\frac{D(t \approx u) \qquad D(u \approx v)}{D(t \approx v)}$$

If every inference rule is preserved by D in the sense above, this yields a decoding algorithm that is sound and complete.

Unfortunately, there seems to be no definition of D that makes this scheme work. To take a simple example, what should $\mathsf{ifeq}(s, t, u, v) \approx w$ decode to? Looking at equations (IFEQ-ELIM) and (IFEQ-INTRO), we should choose $s \approx t \to u \approx w$, since then (IFEQ-ELIM) decodes to the tautology $x \approx x \to y \approx y$, and (IFEQ-INTRO) correctly decodes to $t \approx u \to v \approx w$. But now take the following inference step from the proof of lemma 6, where $\mathbf{t}$ abbreviates the vector of terms $(\mathsf{U}(\mathsf{b}), \mathsf{true}, \mathsf{Q}(\mathsf{b}), \mathsf{true})$:

$$\frac{\mathsf{Q}(\mathsf{b}) \approx \mathsf{ifeq}(\mathbf{t}) \quad \mathsf{ifeq}(\mathbf{t}) \approx \mathsf{true}}{\mathsf{Q}(\mathsf{b}) \approx \mathsf{true}}$$

Decoding this according to D gives

$$\frac{\mathsf{U}(\mathsf{b}) \approx \mathsf{true} \to \mathsf{Q}(\mathsf{b}) \approx \mathsf{Q}(\mathsf{b}) \quad \mathsf{U}(\mathsf{b}) \approx \mathsf{true} \to \mathsf{Q}(\mathsf{b}) \approx \mathsf{true}}{\mathsf{Q}(\mathsf{b}) \approx \mathsf{true}}$$

which is not a valid inference. Notice that the proof of the first premise uses the step $\mathsf{U}(\mathsf{b}) \approx \mathsf{true}$, which is exactly what we need to prove the conclusion by resolution. Unfortunately, D has transformed the first premise into a tautology, so this crucial fact has been lost.

6.2 A Decoding via Labeled Proofs

The problem is how to design D so that it preserves all information from the encoded equation. We first show how to do this for an equation that contains a single occurrence of ifeq. Call the equation $t \approx u[\mathsf{ifeq}(v_1, v_2, v_3, v_4)]$ and its decoding ϕ. The decoding is different depending on *which axiom* introduced the ifeq symbol:

- If it was (IFEQ-ELIM), then to have used it we must have $v_1 \approx v_2$, and it follows that $\mathsf{ifeq}(v_1, v_2, v_3, v_4) \approx v_3$. Hence we define ϕ as $v_1 \approx v_2 \wedge t \approx u[v_3]$.
- If it was an encoded Horn clause, then we must have $v_1 \approx v_2 \longrightarrow v_3 \approx v_4$. By inspection of (IFEQ-INTRO), we also have $\mathsf{ifeq}(v_1, v_2, v_3, v_4) \approx v_4$. Hence we define ϕ as $(v_1 \approx v_2 \longrightarrow v_3 \approx v_4) \wedge t \approx u[v_4]$.

Now the first premise of our example above decodes to $\mathsf{U(b)} \approx \mathsf{true} \wedge \mathsf{Q(b)} \approx \mathsf{Q(b)}$, the second decodes to $(\mathsf{U(b)} \approx \mathsf{true} \longrightarrow \mathsf{Q(b)} \approx \mathsf{true}) \wedge \mathsf{true} \approx \mathsf{true}$, and the conclusion follows by resolution. This decoding works for any transitivity step $t \approx u[\mathsf{ifeq}(v_1, v_2, v_3, v_4)] \approx w$ involving a single ifeq:

- If both premises used (IFEQ-ELIM) to introduce the ifeq, then we have $t \approx u[v_3] \approx w$. Similarly, if they both used an encoded Horn clause, we have $t \approx u[v_4] \approx w$.
- Otherwise, we have (for example) $v_1 \approx v_2 \wedge t \approx u[v_3]$ from the first premise and $(v_1 \approx v_2 \longrightarrow v_3 \approx v_4) \wedge u[v_4] \approx w$ from the second premise. We resolve to get $v_3 \approx v_4$ and rewrite to get $t \approx u[v_3] \approx u[v_4] \approx w$.

Now we show how to generalize this approach to proofs with an arbitrary number of ifeqs. The idea is to *label* each occurrence of ifeq with what sort of axiom was used to introduce it, and use that labeling to drive the decoding. Our algorithm starts by labeling the axioms: In axiom (IFEQ-ELIM), the ifeq is labeled with ELIM, and in encoded Horn clauses it is labeled with INTRO. For Example 10, this results in

$\mathsf{Axiom\ 0:}\ \mathsf{ifeq}_{\mathrm{ELIM}}(x, x, y, z) \approx y$
$\mathsf{Axiom\ 2:}\ \mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{U}(x), \mathsf{true}, \mathsf{Q}(x), \mathsf{true}) \approx \mathsf{true}$
$\mathsf{Axiom\ 4:}\ \mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{S}(x), \mathsf{true}, \mathsf{P}(x), \mathsf{true}) \approx \mathsf{true}$
$\mathsf{Axiom\ 5:}\ \mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{tuple}(\mathsf{P}(x), \mathsf{Q}(y)), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R}(x, y), \mathsf{true}) \approx \mathsf{true}$

The algorithm then adjusts the proofs of the lemmas and goal so that they use the labeled versions of the axioms, which means that these proofs in turn become labeled. Sometimes this creates a *label clash*: a proof $t \approx u[\mathsf{ifeq}(\ldots)] \approx v$ where the $t \approx u$ step and $u \approx v$ step require different labels for the ifeq. We permit such label clashes.

Example 11. The labeled versions of lemma 6 and goal 1 are as follows:

Lemma 6: $\mathsf{Q(b)} \approx \mathsf{true}$
Proof:
$\mathsf{Q(b)}$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{true}, \mathsf{true}, \mathsf{Q(b)}, \mathsf{true})$
$\approx$ { by axiom 1 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{U(b)}, \mathsf{true}, \mathsf{Q(b)}, \mathsf{true})$
{ label clash }
$\mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{U(b)}, \mathsf{true}, \mathsf{Q(b)}, \mathsf{true})$
$\approx$ { by axiom 2 }
true

Goal 1: $\mathsf{R(a, b)} \approx \mathsf{true}$
Proof:
$\mathsf{R(a, b)}$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
$\approx$ { by axiom 4 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{S(a)}, \mathsf{true}, \mathsf{P(a)}, \mathsf{true}), \mathsf{true}),$
$\quad \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
$\approx$ { by axiom 3 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{true}, \mathsf{true}, \mathsf{P(a)}, \mathsf{true}), \mathsf{true}),$
$\quad \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
{ label clash }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{true}, \mathsf{true}, \mathsf{P(a)}, \mathsf{true}), \mathsf{true}),$
$\quad \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
$\approx$ { by axiom 0 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{P(a)}, \mathsf{true}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
$\approx$ { by lemma 6 }
$\mathsf{ifeq}_{\mathrm{ELIM}}(\mathsf{tuple}(\mathsf{P(a)}, \mathsf{Q(b)}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
{ label clash }
$\mathsf{ifeq}_{\mathrm{INTRO}}(\mathsf{tuple}(\mathsf{P(a)}, \mathsf{Q(b)}), \mathsf{tuple}(\mathsf{true}, \mathsf{true}), \mathsf{R(a, b)}, \mathsf{true})$
$\approx$ { by axiom 5 }
true ■

Formally, we model label clashes by admitting the following inference rule, as well as its symmetric variant, writing $\mathbf{v}$ to abbreviate the term vector (v_1, v_2, v_3, v_4):

$$\frac{t \approx u[\mathsf{ifeq}_{\mathrm{ELIM}}(\mathbf{v})] \quad u[\mathsf{ifeq}_{\mathrm{INTRO}}(\mathbf{v})] \approx w}{t \approx w}\ \text{label clash}$$

We claim that labeled proofs (with some caveats described in Section 6.3) can be decoded to first-order proofs based on rewriting and resolution. The main idea is that *label clashes correspond to resolution steps*. Every label clash becomes a resolution step, and all other inferences are passed through essentially unchanged. We start by defining the ifeq-erasure $E(t)$ of a term t as t with the following replacements made:

- Replace every subterm $\mathsf{ifeq}_{\mathrm{ELIM}}(v_1, v_2, w_1, w_2)$ with w_1.
- Replace every subterm $\mathsf{ifeq}_{\mathrm{INTRO}}(v_1, v_2, w_1, w_2)$ with w_2.

We lift E to formulas and to proofs. Notice that the ifeq-erasure of a proof is a valid equational proof *except* where a label clash occurs: the ifeq erases to a different argument on the two sides of the clash.

Next we define a decoding function D that takes a labeled equation and produces a set of Horn clauses; the meaning of an equation is the conjunction of this set. $D(t \approx u)$ consists of the following clauses:

- $E(t) \approx E(u)$;

- for every subterm of the form $\mathsf{ifeq}_{\text{ELIM}}(v_1, v_2, w_1, w_2)$, the clause $E(v_1) \approx E(v_2)$;
- for every subterm of the form $\mathsf{ifeq}_{\text{INTRO}}(v_1, v_2, w_1, w_2)$, the clause $E(v_1) \approx E(v_2) \to E(w_1) \approx E(w_2)$.

For example, for the label clash in lemma 6, we get

- $D(\mathsf{Q(b)} \approx \mathsf{ifeq}_{\text{ELIM}}(\mathsf{U(b)}, \mathsf{true}, \mathsf{Q(b)}, \mathsf{true})) = \{\mathsf{Q(b)} \approx \mathsf{Q(b)}, \mathsf{U(b)} \approx \mathsf{true}\}$ and
- $D(\mathsf{ifeq}_{\text{INTRO}}(\mathsf{U(b)}, \mathsf{true}, \mathsf{Q(b)}, \mathsf{true}) \approx \mathsf{true}) = \{\mathsf{true} \approx \mathsf{true}, \mathsf{U(b)} \approx \mathsf{true} \to \mathsf{Q(b)} \approx \mathsf{true}\}$

from which $\mathsf{Q(b)} \approx \mathsf{true}$ follows by resolution, as desired.

What remains is to decode the proofs. We start with the label clash rule. We must show that the following inference rule is admissible (abbreviating $\mathbf{v} = (v_1, v_2, v_3, v_4)$):

$$\frac{D(t \approx u[\mathsf{ifeq}_{\text{ELIM}}(\mathbf{v})]) \quad D(u[\mathsf{ifeq}_{\text{INTRO}}(\mathbf{v})] \approx w)}{D(t \approx w)}$$

The premises indeed entail $D(t \approx w)$, as the following argument shows. It suffices to prove $E(t) \approx E(w)$, since all other clauses of $D(t \approx w)$ are found in one of the premises: those involving t in the first premise, and those involving w in the second premise. $E(t) \approx E(w)$ can be proved as follows:

- From the first premise, we get $E(v_1) \approx E(v_2)$, and from the second, we get $E(v_1) \approx E(v_2) \to E(v_3) \approx E(v_4)$. Resolution gives $E(v_3) \approx E(v_4)$.
- From the first premise, we get $E(t) \approx E(u[v_3])$, and from the second, we get $E(u[v_4]) \approx E(w)$. Rewriting gives $E(t) \approx E(u[v_3]) \approx E(u[v_4]) \approx E(w)$.

To decode a label clash step, we simply construct this proof.

6.3 A Complication with Congruence

The remaining inference rules are those for equational logic. None of these induce resolution steps and, for the most part, an equational rewrite in the encoded proof turns into a rewrite or in some cases a set of parallel rewrites in the decoded proof. The main bookkeeping required is to keep track of which positions in $t \approx u$ correspond to which positions in $D(t \approx u)$, in order to apply rewrites that occur under the first two arguments of an ifeq.

There is one serious complication, and that has to do with the congruence rule. Suppose that we have the Horn clauses $t \approx t'$ and $t' \approx u \to v \approx w$, and consider the following labeled proof: $\mathsf{ifeq}_{\text{INTRO}}(t, u, v, w) \approx \mathsf{ifeq}_{\text{INTRO}}(t', u, v, w) \approx w$. This should decode to a proof of $t \approx u \to v \approx w$. The first step of the proof uses the congruence rule

$$\frac{t \approx t'}{\mathsf{ifeq}_{\text{INTRO}}(t, u, v, w) \approx \mathsf{ifeq}_{\text{INTRO}}(t', u, v, w)}$$

But this step cannot be decoded: The conclusion decodes to $w \approx w \wedge (t \approx u \to v \approx w) \wedge (t' \approx u \to v \approx w)$, but we have no justification for the two implications.

We believe that this can be solved by a more refined labeling scheme, where an ifeq that occurs only under a context would be labeled as inactive and decoded differently. For now, Twee instead uses an incomplete heuristic approach. Twee proves 907 non-purely-equational Horn TPTP problems (Section 7), 195 of them FOF, and this approach succeeds on all but two. Twee's approach has two parts. First, it handles congruence of ifeq lazily, only applying it in conjunction with transitivity or the label clash rule. In the example above, when decoding the first step, it will produce a proof of $t \approx t'$ and ignore the $\mathsf{ifeq}_{\text{INTRO}}$ context. Second, it uses context from neighboring steps to help with constructing proofs. In the example, once Twee gets to the second step of the proof, it applies the congruence rule to the first step to get $\mathsf{ifeq}_{\text{INTRO}}(t, u, v, w) \approx \mathsf{ifeq}_{\text{INTRO}}(t', u, v, w)$. To do this, it needs to prove the two implications above, which it does by using the proof of $t' \approx u \to v \approx w$ from the second step.

7 Evaluation

We evaluated Tälja on three categories of the TPTP v9.2.1 benchmark suite [38]: HNE (Horn, no equality), HEQ (Horn, with equality), and UEQ (unit equality). These categories contain 821, 452, and 1140 problems respectively, all in CNF. We also evaluated Tälja on the 408 FOF and 7 arithmetic-free TFF theorems of the suite whose axioms and conjecture are Horn clauses as stated. For each problem, we ran E, Twee, and Vampire with a 60-second timeout. E was run with its strategy schedule on one core; Vampire was run with AVATAR, its clause-splitting architecture, disabled; Twee was run with its default settings apart from the tuple encoding of Section 6 (option `--multi`). It does not support TFF. When a prover found a proof, the TSTP output was passed to Tälja, which was run with a 300-second timeout. Tälja, the benchmarking scripts, and the evaluation data are available as an artifact accompanying this paper.

The results are shown in Table 1. "Proved" is the number of problems each prover solved within the time limit. "Translated" is the number of proved problems for which Tälja produced a complete proof within the time limit.

In total, of the three proofs that Tälja did not translate, one exceeded the 300-second translation timeout. It is a Vampire refutation of about a million resolution steps. In the remaining two cases, Twee's proof output component failed.

To validate the proofs, we checked the Lean [13] files output by Tälja. Each step is an explicit proof term, such as an axiom applied to its premises or a congruence for a rewrite. All translated proofs were accepted by the Lean kernel. Tälja can also output a proof as a TSTP derivation, which we checked with GDV [37] for all translated FOF proofs. GDV verified all of them.

8 Related Work

The main inspiration for our proof format and translation algorithm is Denzinger and Schulz's [15] work on the DISCOUNT prover. Its output format is based on lemmas and

Table 1. Evaluation of Tälja on TPTP benchmarks

Set	Prover	Proved	Translated
HNE	E	757	757
	Twee	490	490
	Vampire	722	721
HEQ	E	386	386
	Twee	222	221
	Vampire	360	360
UEQ	E	831	831
	Twee	832	832
	Vampire	736	736
FOF	E	290	290
	Twee	195	194
	Vampire	244	244
TFF	E	6	6
	Vampire	6	6
Total	E	2270	2270
	Twee	1739	1737
	Vampire	2068	2067

equational chains and was adopted by Waldmeister and Twee. Denzinger and Schulz also designed sophisticated heuristics for determining which proof steps should be turned into lemmas. A simple criterion, employed in Twee, is to turn steps that are referenced more than once into lemmas and only those.

In the context of first-order logic without equality, the question of generating readable proofs has been answered multiple times, often with natural deduction as the target. Natural deduction is widely recognized as human-readable, although derivations tend to be very detailed. Miller and Felty [27] gave an early account of such a translation, representing natural deduction proofs as typed λ-terms whose types come from the corresponding resolution refutations. Lingenfelder [25] designed a translation from refutation graph proofs, which can be obtained from resolution proofs, to natural deduction. Huang [21, 22] introduced the notion of assertions to compress natural deduction proof patterns that humans would consider trivial into single steps. Derivations in this system mostly consist of the application of definitions or lemmas. Huang presented two translations, one from standard natural deduction proofs to assertion-level proofs [21], and another from resolution proofs to assertion-level proofs [22]. This line of research culminated with Meier's TRAMP proof translator [26], which implements a variant of Huang's second translation.

For propositional logic, Amjad's [2] translation converted SAT solver resolution-based refutations into natural deduction proofs in linear time, handling refutations with millions of inference steps and producing derivations that could be checked faster than the original refutations.

Orthogonally, De Nivelle [14] designed a translation that transforms resolution proofs containing Skolem symbols to natural deduction proofs without Skolem symbols. This approach is useful to reconstruct proofs in proof assistants that do not include choice axioms; it can also increase the proofs' readability.

In higher-order logic, Andrews [3] designed a translation from refutation matings, which are related to the connection calculus and hence tableaux, to natural deduction proofs. The TPS prover [4] applies this translation to the matings it finds to provide more readable justifications to its users.

Relatedly, the GAPT system [18] was designed to reconstruct a readable proof from an ATP proof using its *expansion proof*, the record of the ground instances used for each quantifier, via a tableau-style search. From there, Ebner and Schlaipfer [19] turned such proofs into natural deduction, often without needing excluded middle. The same ground instances were used by Ebner et al. [17] to find quantified lemmas. Finally, Ebner [16] adapted the search to intuitionistic logic and proved that it always succeeds on Horn problems.

Šinkarovs and Rawson [34] connect the Agda proof assistant to Vampire. On a logical fragment shared by both, Agda goals are passed to Vampire, and the classical proofs it finds are turned into constructive proof terms that Agda type-checks.

Logicians have also studied proof translations for a different reason, asking how they affect proof size rather than readability. Cook and Reckhow's [12] notion of polynomial simulation lets one propositional proof system be judged more powerful than another based on how compactly it can express the same proofs. Nordström's [28] survey traces the consequences of this idea for SAT solving, connecting size lower bounds for resolution proofs to hard limits on how fast resolution-based solvers can run.

9 Conclusion

We introduced a format for capturing resolution and superposition proofs of problems in the Horn fragment of first-order logic. A proof in this format consists of a list of axioms and lemmas ending with the goals. Each lemma or goal is proved either by a chain of equalities or by a "have/hence" block performing hyperresolution steps. Such proofs are arguably more readable than the cryptic proofs by contradiction generated by resolution and superposition provers.

In addition, we designed a translation algorithm from resolution and superposition proofs to our format. We implemented it as a new tool, Tälja, connected it to E and Vampire, and extended Twee's output to interface with it. An evaluation on a variety of TPTP benchmarks showed that our tool translates nearly every proof the three provers find.

Our approach's main limitation is its restriction to Horn clauses. As future work, we want to extend the approach to non-Horn first-order problems and to higher-order logic. We would also like to generate calculation chains for binary predicates other than equality, such as inequalities, as is customary in pen-and-paper mathematics. We also plan to combine Tälja with Krympa [23], a tool for minimizing equational proofs, to produce shorter equality chains.

Acknowledgments

We thank Julius Bernert, Gabriel Ebner, Jan Johannsen, Stephan Schulz, Helmut Schwichtenberg, and Geoff Sutcliffe for suggesting several of the references mentioned in Section 8. We thank Michael Rawson for helpful discussions. We thank Moritz Roos for testing the artifact.

This research was cofunded by the European Union (ERC, Nekoka, 101083038), by the Deutsche Forschungsgemeinschaft (DFG, ConVeY, 378803395), and by the Swedish Research Council (VR, Semantically-guided theorem proving for mathematics, 2025-06153). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Jesse Alama. 2012. Escape to Mizar for ATPs. *CoRR* abs/1204.6615 (2012). arXiv:1204.6615 doi:10.48550/arXiv.1204.6615

[2] Hasan Amjad. 2007. A Compressing Translation from Propositional Resolution to Natural Deduction. In *FroCoS 2007 (LNCS, Vol. 4720)*, Boris Konev and Frank Wolter (Eds.). Springer, 88–102. doi:10.1007/978-3-540-74621-8_6

[3] Peter B. Andrews. 1980. Transforming Matings into Natural Deduction Proofs. In *CADE-5 (LNCS, Vol. 87)*, Wolfgang Bibel and Robert A. Kowalski (Eds.). Springer, 281–292. doi:10.1007/3-540-10009-1_22

[4] Peter B. Andrews and Chad E. Brown. 2006. TPS: A Hybrid Automatic-Interactive System for Developing Proofs. *J. Appl. Log.* 4, 4 (2006), 367–395. doi:10.1016/J.JAL.2005.10.002

[5] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. 1995. DISCOUNT: A System for Distributed Equational Deduction. In *RTA-95 (LNCS, Vol. 914)*, Jieh Hsiang (Ed.). Springer, 397–402. doi:10.1007/3-540-59200-8_72

[6] Leo Bachmair and Harald Ganzinger. 1994. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.* 4, 3 (1994), 217–247. doi:10.1093/LOGCOM/4.3.217

[7] Leo Bachmair and Harald Ganzinger. 2001. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, John Alan Robinson and Andrei Voronkov (Eds.). Vol. I. Elsevier and MIT Press, 19–99. doi:10.1016/B978-044450813-3/50004-7

[8] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. 2023. Superposition for Higher-Order Logic. *J. Autom. Reason.* 67, 1 (2023), article number 10. doi:10.1007/S10817-022-09649-9

[9] Ahmed Bhayat and Martin Suda. 2024. A Higher-Order Vampire (Short Paper). In *IJCAR 2024 (LNCS, Vol. 14739)*, Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt (Eds.). Springer, 75–85. doi:10.1007/978-3-031-63498-7_5

[10] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka, and Albert Steckermeier. 2016. Semi-intelligible Isar Proofs from Machine-Generated Proofs. *J. Autom. Reason.* 56, 2 (2016), 155–200. doi:10.1007/S10817-015-9335-3

[11] Koen Claessen and Nicholas Smallbone. 2018. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In *IJCAR 2018 (LNCS, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 388–404. doi:10.1007/978-3-319-94205-6_26

[12] Stephen A. Cook and Robert A. Reckhow. 1979. The Relative Efficiency of Propositional Proof Systems. *J. Symb. Log.* 44, 1 (1979), 36–50. doi:10.2307/2273702

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE-25 (LNCS, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. doi:10.1007/978-3-319-21401-6_26

[14] Hans de Nivelle. 2005. Translation of Resolution Proofs into Short First-Order Proofs without Choice Axioms. *Inf. Comput.* 199, 1–2 (2005), 24–54. doi:10.1016/J.IC.2004.10.011

[15] Jörg Denzinger and Stephan Schulz. 1994. Recording, Analyzing and Presenting Distributed Deduction Processes. In *PASCO 1994*, Hoon Hong (Ed.). World Scientific, 114–123. doi:10.1142/9789814533584_0012

[16] Gabriel Ebner. 2019. Herbrand Constructivization for Automated Intuitionistic Theorem Proving. In *TABLEAUX 2019 (LNCS, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, 355–373. doi:10.1007/978-3-030-29026-9_20

[17] Gabriel Ebner, Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. 2019. On the Generation of Quantified Lemmas. *J. Autom. Reason.* 63, 1 (2019), 95–126. doi:10.1007/S10817-018-9462-8

[18] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. 2016. System Description: GAPT 2.0. In *IJCAR 2016 (LNCS, Vol. 9706)*, Nicola Olivetti and Ashish Tiwari (Eds.). Springer, 293–301. doi:10.1007/978-3-319-40229-1_20

[19] Gabriel Ebner and Matthias Schlaipfer. 2018. Efficient Translation of Sequent Calculus Proofs Into Natural Deduction Proofs. In *PAAR 2018 (CEUR Workshop Proceedings, Vol. 2162)*, Boris Konev, Josef Urban, and Philipp Rümmer (Eds.). CEUR-WS.org, 17–33.

[20] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. 1997. WALDMEISTER - High-Performance Equational Deduction. *J. Autom. Reason.* 18, 2 (1997), 265–270. doi:10.1023/A:1005872405899

[21] Xiaorong Huang. 1994. Reconstructing Proofs at the Assertion Level. In *CADE-12 (LNCS, Vol. 814)*, Alan Bundy (Ed.). Springer, 738–752. doi:10.1007/3-540-58156-1_53

[22] Xiaorong Huang. 1996. Translating Machine-Generated Resolution Proofs into ND-Proofs at the Assertion Level. In *PRICAI '96 (LNCS, Vol. 1114)*, Norman Y. Foo and Randy Goebel (Eds.). Springer, 399–410. doi:10.1007/3-540-61532-6_34

[23] Lydia Kondylidou, Jasmin Blanchette, and Marijn J. H. Heule. 2026. Tao's Equational Proof Challenge Accepted. In *IJCAR 2026 (LNCS, Vol. 16688)*, Armin Biere, Carsten Lutz, and Sara Negri (Eds.). Springer, 3–21. doi:10.1007/978-3-032-32589-1_1

[24] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *CAV 2013 (LNCS, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 1–35. doi:10.1007/978-3-642-39799-8_1

[25] Christoph Lingenfelder. 1989. Structuring Computer Generated Proofs. In *IJCAI 1989*, N. S. Sridharan (Ed.). Morgan Kaufmann, 378–383. http://ijcai.org/Proceedings/89-1/Papers/060.pdf

[26] Andreas Meier. 2000. System Description: TRAMP: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In *CADE-17 (LNCS, Vol. 1831)*, David A. McAllester (Ed.). Springer, 460–464. doi:10.1007/10721959_37

[27] Dale Miller and Amy P. Felty. 1986. An Integration of Resolution and Natural Deduction Theorem Proving. In *AAAI 1986*, Tom Kehler (Ed.). Morgan Kaufmann, 198–202. http://www.aaai.org/Library/AAAI/1986/aaai86-032.php

[28] Jakob Nordström. 2015. On the interplay between proof complexity and SAT solving. *ACM SIGLOG News* 2, 3 (2015), 19–44. doi:10.1145/2815493.2815497

[29] George A. Robinson and Lawrence Wos. 1983. Paramodulation and Theorem-Proving in First-Order Theories with Equality. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, Jörg H. Siekmann and Graham Wrightson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–313. doi:10.1007/978-3-642-81955-1_19

[30] John Alan Robinson. 1965. Automatic Deduction with Hyper-Resolution. *International Journal of Computer Mathematics* 1, 3 (1965), 227–234. doi:10.1080/00207166508803260

[31] John Alan Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (1965), 23–41. doi:10.1145/321250.321253

[32] Stephan Schulz. 2002. E—a Brainiac Theorem Prover. *AI Commun.* 15, 2-3 (2002), 111–126. http://content.iospress.com/articles/ai-communications/aic260

[33] Stephan Schulz and Adam Pease. 2020. Teaching Automated Theorem Proving by Example: PyRes 1.2 (System Description). In *IJCAR 2020 (LNCS, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 158–166. doi:10.1007/978-3-030-51054-1_9

[34] Artjoms Šinkarovs and Michael Rawson. 2026. When Agda met Vampire. *CoRR* abs/2602.18844 (2026). arXiv:2602.18844 doi:10.48550/ARXIV.2602.18844

[35] Nicholas Smallbone. 2021. Twee: An Equational Theorem Prover. In *CADE-28 (LNCS, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 602–613. doi:10.1007/978-3-030-79876-5_35

[36] Alexander Steen and Christoph Benzmüller. 2021. Extensional Higher-Order Paramodulation in Leo-III. *J. Autom. Reason.* 65, 6 (2021), 775–807. doi:10.1007/S10817-021-09588-X

[37] Geoff Sutcliffe. 2006. Semantic Derivation Verification: Techniques and Implementation. *Int. J. Artif. Intell. Tools* 15, 6 (2006), 1053–1070. doi:10.1142/S0218213006003119

[38] Geoff Sutcliffe. 2024. Stepping Stones in the TPTP World. In *IJCAR 2024 (LNCS, Vol. 14739)*, Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt (Eds.). Springer, 30–50. doi:10.1007/978-3-031-63498-7_3

[39] Markus Wenzel. 1999. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In *TPHOLs 1999 (LNCS, Vol. 1690)*, Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (Eds.). Springer, 167–183. doi:10.1007/3-540-48256-3_12