petar vukmirović

implementation of higher-order superposition

VRIJE UNIVERSITEIT

Implementation of Higher-Order Superposition

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op dinsdag 18 oktober 2022 om 13.45 uur
in een bijeenkomst van de universiteit,
De Boelelaan 1105

door
Petar Vukmirović
geboren te Belgrado, Servië

promotor:	prof. dr. W. J. Fokkink
copromotoren:	dr. J. C. Blanchette prof. dr. S. Schulz
promotiecommissie:	prof. dr. M. Hoogendoorn dr. K. Korovin dr. R. Piskac dr. A. Steen dr. M. Suda prof. dr. G. Sutcliffe

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: automatic theorem proving, higher-order logic, unification

Printed by: Groenprint.nl

Cover: Marina Abramović, Rhythm 10 (The Biography), Cibachrome Print, 1997, © Marina Abramović, Courtesy of the Marina Abramović Archives

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus Serif, Inter and Inconsolata fonts.

Contents

Summary

In the last decades, proof assistants have been immeasurably useful in formally proving validity of hard mathematical theories and correctness of safety-critical software. Using these tools one can formally describe a problem and produce machine-checkable proofs for statements about it. To make the checking phase efficient and trustworthy, proof steps need to be of fine granularity. As a consequence, seemingly simple statements must be proven in minute detail, making the use of proof assistants very tedious.

In an attempt to automate significant parts of this proving process, assistants can invoke automatic theorem provers to finish the proof. However, most assistants are based on higher-order logic, while most automatic theorem provers are based on first-order logic. This means that the two systems must communicate through translations, which obfuscate the conjecture being proved and likely make proving the conjecture more difficult.

In this thesis, we try to bridge this translation gap. We start from E, one of the best first-order proof assistants backends, based on the superposition calculus, and gradually extend it to higher-order logic. As this approach is large in scope, we split it into three parts.

The first part extends E to support a fragment of higher-order logic devoid of λ-abstraction, yielding a prover called Ehoh. Despite this extension being small in scope, Ehoh shows a stronger performance on proof assistant benchmarks than E.

The second part concerns adding support for λ-abstraction. The most important challenge we faced during this extension is that of higher-order unification. At the time we started, the state-of-the-art procedure for full higher-order unification was the one developed in 1970s. We designed a new procedure inspired by this one that enumerates fewer redundant unifiers, has many modern optimizations built in, and can easily be customized to trade its completeness for performance. It is implemented in a prototype prover called Zipperposition, a less efficient but more easily extendible prover compared to E. Our evaluation shows that our procedure substantially improves on the state of the art.

The last part concerns adding native support for Boolean terms. For this part we took inspiration from traditional automatic higher-order provers, based on tableaux. We fitted those techniques in the superposition context and implemented them in Zipperposition. They made Zipperposition the best prover in the higher-order division of the annual CASC theorem prover competition for two consecutive years.

After testing out our ideas in Zipperposition, we decided to go back to Ehoh and extend it to full higher-order logic, obtaining a prover called λE. We chose the most successful approaches implemented in Zipperposition that can be ported easily to λE. The results were once again positive: Our evaluation shows that λE is the best prover on proof assistant benchmarks, and second only to Zipperposition on a standard benchmark set.

We took our idea of porting techniques from weaker to stronger logics one step back: We explored SAT simplification techniques that can be implemented in the superposition

context. We discovered that while some of them can be used during proving, most of them work best as preprocessors.

In conclusion, the work described in this thesis shows that first- and higher-order provers are much more alike than previously thought. Furthermore, we showed that through carefully designed and tuned extension, a first-order prover can become an award-winning higher-order prover.

Samenvatting

In de afgelopen decennia zijn bewijsassistenten bijzonder nuttig gebleken in het formeel bewijzen van de validiteit van complexe wiskundige theorieën zowel als de correctheid van veiligheidskritieke software. Met behulp van deze tools kan men een probleem formeel beschrijven en machinaal controleerbare bewijzen voor uitspraken hierover produceren. Om de controle-fase efficiënt en betrouwbaar te maken, dienen bewijsstappen van een zeer fijne granulariteit te zijn. Als gevolg hiervan moeten op het eerste gezicht triviale feiten in uiterst precies detail bewezen worden, wat het werken met bewijsassistenten buitengewoon omslachtig maakt.

In een poging om belangrijke delen van dit bewijsproces te automatiseren, kunnen bewijsassistenten automatische bewijzers aanroepen om een bewijs af te maken. De meeste bewijsassistenten zijn echter op hogere-orde logica gebaseerd, terwijl de meeste automatische bewijzers op eerste-orde logica gestoeld zijn. Dit betekent dat de twee systemen moeten communiceren door vertalingen, die het probleem dat wordt bewezen vertroebelen en het bewijzen ervan bemoeilijken.

In dit proefschrift proberen we deze vertaalkloof te overbruggen. We beginnen bij E, een van de beste eerste-orde bewijsassistenten backends, gebaseerd op de superpositie calculus, en we breiden hem geleidelijk uit tot logica's van hogere orde. Omdat deze aanpak een omvangrijke reikwijdte heeft, splitsen we hem in drie delen.

Het eerste deel breidt E uit om een fragment van hogere-orde logica te ondersteunen zonder λ-abstractie, wat een bewijzer oplevert die Ehoh heet. Alhoewel deze uitbreiding klein van omvang is, laat Ehoh betere prestaties zien op benchmarks voor bewijsassistenten dan E.

Het tweede deel betreft het toevoegen van ondersteuning voor λ-abstractie. De belangrijkste uitdaging waarmee we tijdens deze uitbreiding werden geconfronteerd is die van hogere-orde unificatie. Op het moment dat we begonnen was de state-of-the-art procedure voor volledige hogere-orde unificatie in de jaren zeventig ontwikkeld. We ontwierpen hierop geïnspireerd een procedure die minder redundante unifiers vindt, waarbij ook veel moderne optimalisaties zijn ingebouwd. Hij kan verder eenvoudig worden aangepast om niet alle unifiers te vinden, maar sneller te zijn. Hij is geïmplementeerd in een prototype prover genaamd Zipperposition, een minder efficiënte maar gemakkelijker uitbreidbare prover, vergeleken met E. Uit onze evaluatie blijkt dat onze procedure een aanzienlijk verbetering oplevert in prestaties.

Het laatste deel betreft het toevoegen van bestaande ondersteuning voor Boolse termen. Voor dit deel hebben we ons laten inspireren door traditionele automatische hogere-orde bewijzers, gebaseerd op tableaus. Die technieken hebben we in de superpositie context toegepast en in Zipperposition geïmplementeerd. Dit maakte Zipperposition in twee opvolgende jaren tot de beste bewijzer in de hogere-orde divisie van de jaarlijkse CASC automatische bewijzers competitie.

Na het testen van onze ideeën in Zipperposition, besloten we terug te gaan naar Ehoh en die uit te breiden tot volledige logica van hogere orde. Dit heeft geleid tot een bewijzer genaamd λE. We hebben hiervoor de meest succesvolle methoden van Zipperposition gekozen die gemakkelijk kunnen worden overgezet naar λE. De resultaten waren wederom positief: Onze evaluatie toont aan dat λE de beste prover is op een benchmark voor bewijsassistenten, en tweede na Zipperposition op een standaard benchmark verzameling.

We hebben vervolgens een stap terug gezet bij onze intentie om technieken van zwakkere naar sterkere logica's over te hevelen: We onderzochten SAT-vereenvoudigingstechnieken die kunnen worden geïmplementeerd in de superpositie context. We ontdekten dat alhoewel sommige ervan kunnen worden gebruikt tijdens het bewijzen, de meeste het beste werken als preprocessors.

Concluderend laat het werk beschreven in dit proefschrift zien dat eerste- en hogereorde bewijzers veel meer op elkaar lijken dan eerder werd gedacht. Verder hebben we laten zien dat door een zorgvuldig ontworpen en afgestemde uitbreiding, een eerste-orde bewijzer een bekroonde bewijzer voor hogere orde kan worden.

Preface

When I downloaded the LaTeX template to write this thesis, the following sentence was present as a placeholder:

> Without a doubt, the acknowledgments are the most widely and most eagerly read part of any thesis.

I do not know if that is true in general, but I have to admit I am guilty of spending quite some time reading acknowledgments. I find that the reason why we spend so much time reading acknowledgments is that we want to peek behind the rigorous, formal shape that a PhD thesis is required to take. We also want to see what PhD candidates have to say about their personal experiences of finishing a PhD.

After reading many preface or acknowledgments chapters I realized that there is a discrepancy between what research shows candidate's experience of finishing a PhD is and what is actually written in these parts of theses. An article by Levecque et al. [104] states that one in two PhD students experiences psychological distress, while one in three is at risk of a common psychiatric disorder. In light of this fact, I want to write this chapter not only to the people whose help and work indebted me, but also to (future) PhD students that, after looking up a detail in this thesis, voyeuristically read this chapter (like I did many times for other theses).

In my first year of PhD, I was overwhelmed with fear and doubts. Did I make the right choice by starting a PhD? Am I performing up to expectations? How many papers should I publish? Should I read more papers? Did I *actually* learn anything? I could go on an on like this.

The fact that I was surrounded by very successful postdocs and PhD students in later years of their studies made me feel like I was the only one having this problem. To the students reading this chapter, I would very much like to assure you that you are not.

What helped me get out of the self-doubting loop was actually taking action. From the simplest ones, like saying "I don't understand" out loud instead of silently nodding when I had trouble following scientific discussion, to the ones with more impact on my career like actively looking for projects I could join while I did not have much work on my plate. As the years went on I felt that I could more openly talk about my insecurities and I found out that many people in my situation have them as well.

What made my PhD journey more enjoyable are undoubtedly the people I worked with. I want to thank the whole Theoretical Computer Science group at the VU which was there with me from the early days of my master studies. They helped me organize my master programme so that I follow more theory courses which most likely led me to start an academic career. I also want to thank other postdocs and PhDs that studied during my PhD or are still studying. Most notably, I would like to thank Alexander Bentkamp, one of the most gifted mathematicians I know. I coauthored many of my papers with him, each of which was a pleasure to work on. I would also like to thank all of my collaborators and

people that suggested improvements to the articles I wrote. Notably, I owe debt of thanks to the committee members who carefully read this thesis and contributed to it by their helpful comments.

I want to thank my daily supervisor, Jasmin Blanchette. He always says that the main product of a PhD is not a thesis but the person you come out as after four years. With his guidance, I learned to approach things more rigorously, slower, and with more attention. He also helped me tremendously with my writing and presentation skills. Even though he tried very hard, he did not teach me to use articles (i.e., English articles "the", "a", or "an") though, and capitulated by adding them to many of the articles I wrote.

My other two supervisors also helped me greatly. Wan Fokkink read the thesis very carefully and provided very valuable comments. For the last four years he made my PhD a smooth sail by solving all the administrative issues on time and provided a very nice work environment. Stephan Schulz is a real programming and logic wizard. His deep understanding of all things superposition and low-level Linux programming helped me not only with my PhD, but I also learned a lot of new skills that are undoubtedly useful for my future career.

Predrag Janičić, my professor from University of Belgrade who inspired me to deepen my knowledge in various areas of computer science, suggested many improvements to the text of this thesis. I thank him as well.

Lastly, I would like to thank Martijn. He was besides me for the last four years and helped me overcome many of the doubts and issues I mentioned. He also always patiently waits until my medication or second glass of Grüner Veltliner kicks in (not at the same time). I also want to thank Jelisaveta and Tara, which sometimes had to deal with me when medication or wine did not kick in. All my other friends, in Serbia or the Netherlands, thank you for your support.

Petar
Amsterdam, April 2022

1 Introduction

Half sleeping, half playing Fruit Ninja on my smartphone at one of the philosophy lectures in my high school I heard my professor exclaim with excitement: "Mathematical truth is the highest form of truth!". Unbothered and uninterested, I continued playing Fruit Ninja.

Many years later I started studying various formal methods in computer science. There I learned the value of formal, mathematical language and finally understood it. Not only does using a rigorous mathematical language avoid the ambiguity of the natural one, it allows us to use the formal rules of reasoning to make conclusions from initial premises in a trustworthy manner.

Though the interest for rigorous and formal reasoning dates back to Aristotle, it intensified at the end of 19th and the beginning of 20th century [62]. Researchers in that time were mostly interested in finding a formal language that is expressive enough to describe many mathematical theories, yet intuitive and understandable enough to allow reasonably simple formal reasoning. First-order logic became a commonly used system that satisfies both requirements. This formalism not only allows one to model simple logical relations such as "if it rains, the road is muddy" (or formally $\mathsf{rains} \longrightarrow \mathsf{muddy}$), but also to quantify over objects as in "all people are mortal" ($\forall x.\, \mathsf{human}(x) \longrightarrow \mathsf{mortal}(x)$).

Automatic Theorem Proving The initial study of first-order logic and the rules to reason about statements in it was prolific. In 1930 Gödel [72] showed that there is a system of inference rules (calculus) such that for any valid statement (i.e., a theorem) a proof can be constructed in this system. Around the same time, the first computers were leaving the imagination of the engineers and began automating many complicated processes. Naturally, the question as to whether a computer can decide if a first-order statement is valid arose. Church [47] and Turing [164] answered this question negatively in 1936.

Despite this negative result, the prospect of automatically proving theorems of first-order logic remained too enticing. In 1960, Davis and Putnam [52] described a procedure for checking validity of a first-order formula. This procedure terminated only on valid formulas, as Church and Turing proved no procedure can check validity of an arbitrary (possibly invalid) formula in finite time. Even though it was efficient enough to prove only simple formulas, it was an impressive achievement for the time.

1

A bigger breakthrough happened in 1965 when Robinson introduced a calculus that would shape the future of automatic reasoning — the resolution calculus [142]. Consisting of a single inference rule it was simple and elegant. It was also *complete*: each theorem could be proven using this system. Furthermore, it was less explosive than Davis and Putnam's algorithm since it provided better guidance for search space exploration.

Since its introduction, resolution was improved in many ways. Strategies and heuristics [175] to curb the explosion of the search space were introduced, as well as complete, but less explosive variants of the calculus [44]. Despite this progress, reasoning about equality of objects was still hard for resolution. Even though some improvements, like paramodulation calculus [123], improved this situation, reasoning with equality remained impractical. This changed with introduction of superposition [7] in the early 1990s.

Superposition is a complete calculus for first-order logic loosely based on resolution, featuring support for equality on the calculus level. This support is further optimized by use of various heuristics built to help prune the search space. Thanks to efficient implementation, this calculus was used to prove long-standing open problem known as the Robbins conjecture [114]. Provers based on this calculus have been winning the first-order division of the CASC theorem proving competition since its inception in the 1990s [163].

Even though the successes of proving the Robbins conjecture showed that it is possible, automatically proving hard mathematical problems is still out of reach for theorem provers. However, there are other ways in which humans and computers can work together to construct formal proofs.

Proof Assistants and Hammers Proof assistants (also called interactive theorem provers) are programs which allow users to describe a theory formally and to write proofs of the statements about this theory using inference rules of the proof assistant. These programs can then check if the given proof is correct. As their core is usually small and well-tested, proofs that pass the checking phase are considered trustworthy.

Proof assistants were essential in proving very complex mathematical theorems in a trustworthy manner. For example, the four-color theorem [71] and the Kepler conjecture [73] were proven inside proof assistants, ending the need to trust hundreds of pages of mathematical proofs. Proof assistants are also used to show correctness of complex software such as compilers [103] and operating systems kernels [94]. This verified software can be used in safety-critical scenarios such as nuclear plants and autonomous vehicles.

The proofs need to be spelled out in minute detail to be accepted by a proof assistant. Even though many proof assistants offer some sort of automation, seemingly simple statements cannot be automatically proved. This is one of the reasons why large verification projects can take years to finish. Thus, providing some level of proof automation can substantially increase productivity of users of proof assistants.

Integrating proof automation directly into the proof assistant is not an easy task. Most proof assistants are based on variants of higher-order logic, a logic expressive enough to describe complex mathematical theories out of reach for first-order logic. However, it is harder to automate as in general it does not even allow complete proof calculi. To provide some level of automation, proof assistant developers use the efficient first-order provers as follows. First, the proof goal is translated from higher-order to first-order logic. Then the first-order prover is run on the translated problem. If the first-order prover finds the proof,

the proof is reconstructed in the proof assistant. Modern assistants such as Coq [23], HOL Light [74], and Isabelle [130] implement the approach using the *hammers* CoqHammer [51], HOL(y)Hammer [89], and Sledgehammer [34], respectively.

The approach proved useful: on some benchmarks, around 77% of the proof goals can be discharged automatically using Sledgehammer [30]. However, there is clearly some untapped potential in the approach as it communicates with automatic theorem provers through a translation. To bridge this translation gap, an attempt was made to use higher-order automatic theorem provers instead of first-order ones [156]. This proved unfruitful as state-of-the-art higher-order theorem provers at the time of this attempt were primarily designed for small problems requiring complex higher-order reasoning. On the other hand, problems coming from hammers consist of mostly first-order formulas, are rather large, and mostly require simple higher-order proof steps. An example of a problem that is seemingly simple, but whose translation to first-order logic is out of reach for even the best of first-order provers is [16]:

$$\left(\sum_{i=1}^{n} i^2 + 2i + 1\right) \approx \left(\sum_{i=1}^{n} i^2\right) + \left(\sum_{i=1}^{n} 2i\right) + \left(\sum_{i=1}^{n} 1\right)$$

One of the reasons why this problem is hard is that higher-order proof assistants rely on unnamed functions, called *λ-abstractions*, to represent variable-binding mathematical operators such as summations, integrals and limits. More precisely, the left-hand side of the above equation is usually represented as $\mathsf{sum}(1, n, \lambda i.\ i * i + 2 * i + 1)$. When translated to first-order logic, λ-abstractions, which are ubiquitous as all-purpose binders in higher-order logic, are obfuscated to the point that all but the simplest reasoning about them is out of reach for first-order provers. However, for higher-order provers the crux of the proof is matching the axiom $\forall i\, j\, f\, g.\, \mathsf{sum}(i, j, \lambda x.\ f\ x + g\ x) \approx (\mathsf{sum}(i, j, \lambda x.\ f\ x) + \mathsf{sum}(i, j, \lambda x.\ g\ x))$ against the goal. This operation is relatively simple and, in general, much more efficient than first-order reasoning about translations.

Thus, from the standpoint of proof assistants the ideal automatic theorem prover fulfills the following wishlist:

1. Performs as the best first-order provers on first-order problems
2. Scales well with the size of the problem
3. Supports higher-order logic and scales with the amount of higher-order axioms

This brings us to the topic of this thesis: *How to develop a prover fulfilling this wishlist?*

Our Approach To improve the automatic reasoning for higher-order logic, my colleagues and me decided not to start from a blank slate and build a new theorem prover. Rather, we start from a position of strength—from the state-of-the-art first-order superposition theorem prover E [147]—and extend it to higher-order logic. In this way, the points 1 and 2 from the above wishlist are already fulfilled, as E is very efficient and deals well with large problems.

We did not extend E to higher-order logic in one atomic step. Instead, we extended it by adding support for features of higher-order logic one by one, which gradually increase

1

its reasoning capabilities. With this gradual approach we made sure that after every extension, E's efficiency on first-order problems did not decrease. Thus, while trying to fulfill the point 3 from the wishlist we made sure that points 1 and 2 were not affected.

We have identified three stops on the road to full higher-order logic. A distinguished feature of higher-order logic is that it not only supports quantification over objects, but also over functions that manipulate these objects. Thus, higher-order logic allows us to make a statement "all functions with arguments 0 and 1 result in 2" ($\forall x.\, x(0,1) \approx 2$). Replacing x with the addition function (+) is enough to disprove this statement. The first stop is to extend E to support quantification over functions that are already present in the original problem.

Higher-order logic also supports expressing functions that are not explicitly present in the original problem. Suppose we want to prove there is a function that returns 4 when given 2 and 5 when given 3 ($\exists x.\, x(2) \approx 4 \wedge x(3) \approx 5$). An example of such function is the function that adds 2 to its argument. In higher-order logic such unnamed functions can be represented using λ-abstractions as $\lambda y.\, y + 2$. This term suffices to prove the original statement. The second stop is to support automatically synthesizing such functions.

In higher-order logic we can make statements about statements. More precisely, it treats Boolean terms (formulas) as first-class citizens, enabling us to model some concepts in a more natural manner. For example, statements such as "what Jasmin says is true" can be represented naturally as $\forall x.\, \mathsf{says}(\mathsf{jasmin}, x) \longrightarrow x$. The last stop is to support Boolean terms as first-class citizens.

1.1 Contributions

The main contribution of this thesis is the extension of two theorem provers, E and Zipperposition [48], to full higher-order logic. To extend these provers, we faced many challenges. Some of them were of engineering nature, some of them were algorithmic, while others concerned fine-tuning the heuristics to optimize the performance. We discuss those challenges in more detail.

First, we implemented the complete superposition calculus for λ-free higher-order logic [15] in E, obtaining a prover we call Ehoh. The name of the prover is a word play which combines one of the hit songs by the Teletubbies[1] and the fact that our prover is a higher-order (HO) extension of E. We had to modify the internal term representation of E to support λ-free higher-order terms, implement new unification and matching algorithms, extend term indexing data structures to work with higher-order terms, and improve the performance of heuristics on higher-order problems.

Alexander Bentkamp with the help of other colleagues and me developed a complete superposition calculus for the second phase described above [18]. Due to the complexity of the calculus and many design decisions that had to be evaluated in a short amount of time we decided to implement the calculus in Zipperposition [48, 49]. This prover is less efficient than E, but it is implemented in OCaml and it was designed to make prototyping ideas and experimenting with calculi much easier. In 2019 we entered the higher-order division of CASC theorem proving competition [160] with Zipperposition. It finished third, one percent point behind Leo-III [153] and 12 percent points behind Satallax [39].

[1] https://en.wikipedia.org/wiki/Teletubbies_say_%22Eh-oh!%22

Through this experience we identified one of the bottlenecks of Zipperposition: its unification algorithm. To remove this bottleneck we developed a complete higher-order unification algorithm that is more efficient and generates fewer redundant unifiers than the state of the art.

We also noticed that Zipperposition's support for first-class Booleans was lacking compared to competition. We have closely studied the problems on which Zipperposition fails but are solved by other provers, and created a set of incomplete rules that enhance Zipperposition's Boolean support. Zipperposition 2 implemented these rules together with the new unification algorithm. In 2020, Zipperposition 2 won the higher-order division of CASC [161]. This time it was 20 percent points ahead of the second best prover, Satallax.

Inspired by this success, we implemented the complete superposition calculus for higher-order logic with Booleans (i.e., the third phase), that was developed by Bentkamp with the help of other colleagues and me [17]. Developing this calculus helped us understand in which ways our pragmatic extension was too weak to prove hard theorems of higher-order logic. We further fine-tuned the heuristics and implemented new ways to tame the search space explosion. In 2021 we released Zipperposition 2.1 which implemented these new features. It again won CASC, outperforming Zipperposition 2. It was 16 percent points ahead of the runner-up, Vampire 4.5 [100].

Building on all the experience we gathered while working on Zipperposition, we further extended Ehoh to full higher-order logic, obtaining a prover called λE. We again modified the term representation, implemented new unification and matching algorithms, extended indexing data structures, and tweaked heuristics. Even though this extension has a larger scope than the original extension from E to Ehoh, the strong basis that we had built earlier made it manageable.

A common thread for the extensions of both E and Zipperposition is that we heavily relied on the availability of established higher-order provers and the TPTP benchmark set. Whenever we noticed that there were problems E and Zipperposition could not solve, but other competitive provers could, we looked for an explanation for this issue. This explanation would usually be in the form of missing inference or heuristic. Then we would try to find more problems that have a similar pattern and see if the developed technique is useful on them. If so, we would make it part of the portfolio of approaches tried on every problem. The ideas for our unification procedure (Chapter 4) and the techniques described in Chapters 5 and 6 were developed in this way.

After spending so much time and energy on developing higher-order provers, we took a step back and worked on improving first-order provers. As they form the basis of higher-order reasoning in our approach, this should improve higher-order performance as well. In particular, we looked at efficient approaches to simplify problems in propositional logic, which is simpler than first-order logic and even decidable. We lifted some of these propositional approaches to first-order logic and implemented them in Zipperposition.

1.2 Thesis Structure and Publications

This thesis is cumulative: chapters 3–8 are taken from previous publications (or drafts in case of Chapter 7) where I was the main author. Use of the publications in this thesis was authorized by the coauthors.

1

Chapter 2 introduces the background material necessary for following this thesis, while Chapter 9 gives a brief summary of the work and describes alleys for future work. The other six chapters discuss previously described contributions in detail:

Chapter 3 discusses the steps we have taken to extend first-order prover E to λ-free higher-order logic, obtaining Ehoh. It is mostly based on the journal publication

1. Petar Vukmirović, Jasmin Blanchette, Simon Cruanes, and Stephan Schulz. Extending a Brainiac Prover to Lambda-Free Higher-Order Logic. In *International Journal on Software Tools for Technology Transfer*. Springer, 2021. Published online only.

while parts of the chapter were previously published in the conference publication

2. Petar Vukmirović, Jasmin Blanchette, Simon Cruanes, and Stephan Schulz. Extending a Brainiac Prover to Lambda-Free Higher-Order Logic. In Vojnar, T., Zhang, L. (eds.) *TACAS 2019*, LNCS 11427, pp. 192–210, Springer, 2019.

Chapter 4 discusses the higher-order unification procedure we designed to be used inside efficient theorem provers. It is mostly based on the journal publication

3. Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient Full Higher-Order Unification. In *Logical Methods in Computer Science* 17(4): 18:1–18:31, 2021.

while parts of the chapter were previously published in the conference publication, which won the best paper by a junior researcher award:

4. Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient Full Higher-Order Unification. In Ariola, Z.M. (ed.) *FSCD 2020* LIPIcs 167, pp. 5:1–5:17, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

Chapter 5 discusses the pragmatic extensions of the complete calculus for superposition with λ-abstractions to support first-class Booleans. It is based on the publication

5. Petar Vukmirović and Visa Nummelin. Boolean Reasoning in a Higher-Order Superposition Prover. In P. Fontaine, K. Korovin, I.S. Kotsireas, P. Rümmer, S. Tourret (eds.) *PAAR-2020*, CEUR Workshop Proceedings, vol. 2752, pp. 148–166. CEUR-WS.org (2020)

Chapter 6 discusses the approach we took to control the explosion of the search space inherent to higher-order superposition. It is mostly based on the journal publication

6. Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. In *Journal of Automated Reasoning*. Springer, 2022.

while parts of the chapter were previously published in the conference publication, which won the best paper by a student award:

7. Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. In Platzer, A., Sutcliffe, G. (eds.) *CADE-28*, LNCS 12699, pp. 415–432, Springer, 2021.

Chapter 7 discusses the extension of Ehoh to support full higher-order logic, resulting in λE. It is based on the draft

8. Petar Vukmirović, Jasmin Blanchette, and Stephan Schulz. Extending a High-Performance Prover to Higher-Order Logic. 2022.

Chapter 8 discusses lifting some of the techniques used to simplify formulas in propositional logic to first-order logic. It is based on the technical report version of the publication:

9. Petar Vukmirović, Jasmin Blanchette, and Marijn J.H. Heule. SAT-Inspired Eliminations for Superposition. In Piskac, R., Whalen, M. (eds.) *FMCAD 2021*, pp. 231–240, IEEE, 2021.

As a PhD candidate, I coauthored the following articles and papers, that are not included in this thesis:

10. Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. Seventeen Provers Under the Hammer. Accepted at ITP 2022.

11. Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with Lambdas. *Journal of Automated Reasoning* 65(7): 893–940, 2021.

12. Visa Nummelin, Alexander Bentkamp, Sophie Tourret, and Petar Vukmirović. Superposition with First-Class Booleans and Inprocessing Clausification. In Platzer, A., Sutcliffe, G. (eds.) *CADE-28*, LNCS 12699, pp. 378–395, Springer, 2021.

13. Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. Superposition for Full Higher-Order Logic. In Platzer, A., Sutcliffe, G. (eds.) *CADE-28*, LNCS 12699, pp. 396–412, Springer, 2021.

14. Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, Higher, Stronger: E 2.3. In Fontaine, P. (ed.) *CADE-27*, LNCS 11716, pp. 495–507, Springer, 2019.

15. Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with Lambdas. In Fontaine, P. (ed.) *CADE-27*, LNCS 11716, pp. 55–73, Springer, 2019.

1.3 Related Work

The earliest attempts to automate higher-order logic can be traced back to Robinson, who provided two approaches. The first one [140] operates directly on higher-order formulas. The second approach [141] is based on translating higher-order logic to first-order logic using combinators, similar to the approach taken by hammers. Huet [81] and Andrews [1] designed calculi that are directly applied to higher-order formulas.

These attempts resulted in early higher-order theorem provers. Andrews and colleagues developed TPS [3]: an automatic prover which allows users to specify proof outlines, based on expansion proofs. Benzmüller and colleagues developed LEO [19], a prover based on higher-order resolution which introduced the cooperative paradigm. Provers implementing this paradigm invoke first-order backends in regular intervals to finish the proof. The latest iteration in the LEO family of provers is Leo-III [153]. Satallax [39] is based on higher-order analytic tableaux. Such representations of logical formulas are tree

structures with at each node a subformula of the original formula to be proved or refuted. Satallax improves on the method by using a SAT solver to close tableau branches. It won the higher-order division of CASC on eight occasions: in 2011 and every year from 2013 to 2019.

Various researchers also experimented with the idea of extending a first-order prover to higher-order logic. Beeson [14] implemented second-order unification and added λ-terms to one of the best first-order provers at the time, Otter [115]. Cruanes extended Zipperposition [48, 49] with arithmetic, induction, and rudimentary support for higher-order logic. Bhayat and Reger implemented a complete superposition calculus for higher-order logic, based on combinators in Vampire [25]. Authors of SMT (satisfiability modulo theories) solvers followed a similar path: they extended cvc4 [12] and veriT [38] to support higher-order logic [11].

Higher-order unification is one of the central procedures in a higher-order prover. Jensen and Pietrzykowski introduced a complete procedure for enumerating elements unifiers for higher-order terms [86]. Huet noticed that some calculi remain complete if solving a difficult subproblem of higher-order unification ("flex-flex" pairs) is delayed. He described a procedure to solve this easier problem in a more efficient way [82]. Dougherty desgined a procedure to enumerate higher-order unifiers using higher-order combinators [56].

To make unification more efficient, Bhayat and Reger implemented efficient term indexing in their extension of Vampire [25]. Libal and Steen designed efficient term indexing based on substitution trees [106].

There have been many studies on the performance of algorithms or success rate of heuristics in first-order theorem proving. For example, Hoder and Voronkov evaluated different unification algorithms [79], while Reger, Suda, and Voronkov evaluated different parameters of a novel prover architecture [136]. Other authors evaluated different proof search heuristics for superposition provers [70, 78, 148]. Evaluation of many prover parameters has been done for higher-order provers as well [21, 60, 152, 174].

2

Preliminaries

In this chapter we lay out the basic prerequisites for the remaining chapters. We begin by describing the three logics that we work with in this thesis: propositional logic, first-order logic, and higher-order logic. Then, we explain the clausal structure which is the backbone of many calculi for automated provers. We finish with the description of the superposition calculus. As this thesis discusses practical aspects of theorem proving, we define only the fundamental notions, while more advanced notions are intuitively described with references to rigorous definitions. The text of this chapter is partly based on the preliminaries sections of the publications listed in Chapter 1.

2.1 Propositional Logic

Atomic formulas (*atoms*) of propositional logic are propositional variables $\mathsf{p}, \mathsf{q}, \ldots$ and constants $\top$ and $\bot$. More complex formulas are built inductively using logical connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$: if ϕ and ψ are formulas, then $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi$ are formulas as well.

To interpret the formulas, propositional variables are assigned 0 (false) or 1 (true), and constants $\top$ and $\bot$ are assigned 1 and 0, respectively; formulas are interpreted using the rules for each connective [84, Sect. 1.4]. As there are finitely many propositional variables in a formula, trying all (finitely many) possible variable assignments describes an algorithm to decide satisfiability (existence of satisfiable assignment for a formula) or validity (if all assignments satisfy the formula).

This simple approach is prohibitively expensive and modern tools that decide the propositional satisfiability problem (SAT solvers) use more advanced approaches such as the CDCL calculus [149]. A crucial improvement of this calculus compared to more naive approaches is that it does not backtrack chronologically when it determines that the partial model constructed for a set of clauses does not satisfy all clauses. Instead, it generates a *learned clause* which explains how this conflicting state was reached, which enables smarter backtracking. Modern SAT solvers also heavily preprocess the problem and continue simplifying it during the proving process.

2.2 First-Order Logic

First-order logic increases the expressivity by allowing quantification over objects and has a more complicated formula structure. There are many flavors of first-order logic, but in this thesis we consider monomorphic first-order logic with equality.

We distinguish a set of base types T which is required to include the Boolean type o, and a set of symbols Σ. To each symbol $\mathsf{f} \in \Sigma$ a tuple $(\tau_1, \ldots, \tau_n, \tau)$, $n \geq 0$ of types is assigned, written $\mathsf{f} : (\tau_1, \ldots, \tau_n) \rightarrow \tau$. We say that $(\tau_1, \ldots, \tau_n)$ are *argument types*, τ is the *return type*, and n is the *arity* of the symbol f. If the return type of f is o, we call f a *predicate symbol*, otherwise we call it a *function symbol*. Argument types may not be Boolean.

Terms are the basic building blocks of first-order logic, built inductively as follows. Variables $x, y, \ldots$, assigned types $\tau \in T \setminus \{o\}$, are terms. If $t_1, \ldots, t_n$ are terms of types $\tau_1, \ldots, \tau_n$, respectively, and $\mathsf{f} : (\tau_1, \ldots, \tau_n) \rightarrow \tau$, then $\mathsf{f}(t_1, \ldots, t_n)$ is a term of type τ. If $n = 0$, we drop the parentheses and write f. We also abbreviate $\mathsf{f}(t_1, \ldots, t_n)$ to $\mathsf{f}(\overline{t}_n)$. Using a similar notation, we abbreviate a tuple of terms $(t_1, \ldots, t_n)$ as $(\overline{t}_n)$ or simply $\overline{t}_n$.

Terms are used to build *atoms*. An atom is either a term t of type o (*predicate atom*) or an equation $s \approx t$ where terms t and s are of the same type (*equational atom*). Atoms are combined using logical connectives to build formulas just like in propositional logic. Additionally, first-order formulas are built using quantifiers: if ϕ is a formula and x is a variable, then $\forall x.\, \phi$, as well as $\exists x.\, \phi$ are formulas. Intuitively, the first formula requires that ϕ holds for every value of x, while the second one requires that there is some x for which ϕ holds. As soon as there is a single functional symbol with arity greater than 0, there are infinitely many terms that can be substituted for a free variable. Furthermore, in first-order logic, it does not suffice to assign values to variables to determine the truth value of the formula. It is also necessary to interpret the symbols. Therefore, it is obvious that the

propositional technique for deciding satisfiability does not work in the first-order case.

Even though we do not formally define the semantics of logic, we assume the natural extensions of domain, valuation, interpretation, and model (as defined by Fitting [65]) from unsorted to many-sorted logic. The models we consider are *normal*, i.e., they interpret $\approx$ as an equality relation. Usual notions of (un)satisfiability and (in)validity are assumed. We write $\mathcal{I} \vDash_\xi N$ to denote that a model $\mathcal{I}$ satisfies a formula set N, for a variable assignment ξ. If $\mathcal{I}$ is a model of N (i.e., $\mathcal{I}$ satisfies it under every variable assignment), we simply write $\mathcal{I} \vDash N$. Overloading notation, we write $M \vDash N$ to denote that M *entails* N, i.e., that every model of M is a model of N.

A *position* in a term is a tuple of natural numbers, with ε denoting the empty tuple. *Subterms* and positions are inductively defined as follows. The term t is a subterm of itself at position ε. If s is a subterm of t_i at position p, then s is a subterm of $\mathsf{f}(\overline{t}_n)$ at position $i.p$ (with . denoting prepending operation). A *context* is a term with zero or more subterms replaced by a hole $\square$. We write $C[\overline{u}_n]$ for the term resulting from filling in the holes of a context C with the terms $\overline{u}_n$, from left to right. We say a term is *ground* if it has no variables. By $u[s]$ we denote that s is a subterm of u.

Substitutions $\sigma, \varrho, \ldots$ are total mappings from variables to terms of the same type. Substitutions map only finitely many variables to terms other than the variable itself. This is denoted as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where x_i are variables that are not mapped to themselves. Applying a substitution σ to a term t, denoted $\sigma(t)$, results in replacing all variables of t by the corresponding values of the mapping σ. The composition $\varrho\sigma$ of substitutions is defined by $(\varrho\sigma)t = \varrho(\sigma(t))$. We call a substitution σ *grounding* if $\sigma(x) \neq x$ implies that $\sigma(x)$ is ground.

2.3 Higher-Order Logic

In this thesis we use classic extensional simply typed monomorphic higher-order logic with the choice operator, but without the description operator, the axiom of infinity, and the axiom of at least two individuals [20, Sect. 3.5]. This logic corresponds to Henkin's extensional type theory. We additionally use some variations of this logic, clearly stating the differences. Assuming a set of base types T and a set of symbols Σ, let us define terms and types. Types are either base types $\tau \in T$, or function types $\tau \to \upsilon$ where both τ and υ are types. Each symbol $\mathsf{f} \in \Sigma$ is assigned a type.

Terms are defined as free variables $F, X, \ldots$, bound variables $x, y, z, \ldots$, or symbols $\mathsf{f}, \mathsf{g}, \mathsf{a}, \mathsf{b}, \ldots$. Furthermore, if s and t are terms of type $\tau \to \upsilon$ and τ, respectively, then $s\,t$ is a term of type υ. Lastly, if x is a bound variable of type τ and s is a term of type υ, then λ-abstraction $\lambda x.\, s$ is a term of type $\tau \to \upsilon$. The syntactic distinction between free and bound variables gives rise to *loose bound variables* [124], bound variables that appear without an enclosing λ binder (e.g., y in $\lambda x.\, y\,\mathsf{a}$). Note that there is no requirement that a symbol is not applied to terms of Boolean type as in first-order logic. Furthermore, all logical connectives are symbols present in the set Σ. This means that higher-order logic does not distinguish between terms and formulas. For example, $\mathsf{p} \wedge \mathsf{q}$ and $\mathsf{f}(\mathsf{p} \wedge \mathsf{q})$ are well-formed terms of higher-order logic. For convenience, we still call terms of Boolean type *formulas*. Our logic also supports the Hilbert choice operator ε. Intuitively, this operator chooses an arbitrary value that satisfies a predicate and it is axiomatized as follows: $\forall p.\, (\exists x.\, p\,x) \to p\,(\varepsilon\, p)$.

With α-conversion we assume the rule $(\lambda x.\, s) \longrightarrow \lambda y.\, \{x \mapsto y\}(s)$ where y does not occur as a loose bound variable in s and y is not bound in s. Application of a substitution to a term implicitly α-renames bound variables to avoid capture. For example, $\{X \mapsto x\}(\lambda x.\, X\,\mathsf{a})$ results in $\lambda y.\, x\,\mathsf{a}$. The β-reduction rule, roughly speaking, corresponds to passing arguments in a function call: $(\lambda x.s)\, t \longrightarrow \{x \mapsto t\}(s)$, where the bound variables in s are renamed to avoid capture. Lastly, η-reduction is defined as $\lambda x.\, s\, x \longrightarrow s$ under the condition that x is not loose bound in x. We write $s \longleftrightarrow^*_{\alpha\beta\eta} t$ if terms s and t are equal modulo $\alpha\beta\eta$-conversion. Terms that are convertible with respect to any of the rules form equivalence classes. Thus, we also say that $\alpha(\beta\eta)$-convertible terms are $\alpha(\beta\eta)$-equivalent.

In practice, many implementations of λ-terms avoid a named representation of bound variables described above by using a *locally nameless representation* [45]. In this representation, free variables retain their names, while abstractions are simply represented by a λ symbol, without any names. Further, bound variables can be represented using *De Bruijn indices*. A bound variable with De Bruijn index $\boldsymbol{i}$ is bound by the $i+1$-th enclosing λ binder. For example, the term $\lambda x.\, \mathsf{f}(\lambda yz.\, \mathsf{g}\, z\, x)$ is represented as $\lambda\, \mathsf{f}(\lambda\lambda\, \mathsf{g}\, \mathbf{0}\, \mathbf{2})$. The locally nameless representation reduces the α-equivalence test to the syntactic equality test.

As hinted in previous definitions, higher-order substitutions $(\sigma, \varrho, \ldots)$ are functions from both free and bound variables to terms. A variable F is mapped by σ if $\sigma(F) \mathrel{\reflectbox{$\not$}\leftrightarrow}^*_{\alpha\beta\eta} F$. Substitutions map only finitely many variables. Given a substitution ϱ, which maps F to s, with $\varrho \setminus \{F \mapsto s\}$ we denote the substitution that does not map F and otherwise coincides with ϱ. Given substitutions ϱ and σ, mapping disjoint variable sets, we write $\varrho \cup \sigma$ to denote $\varrho\sigma$.

We let $s\, \overline{t}_n$ stand for $s\, t_1 \ldots t_n$ and $\lambda\overline{x}_n.\, s$ for $\lambda x_1 \ldots \lambda x_n.\, s$, omitting the length $n \geq 0$ when it is not important or it can be inferred. Every β-reduced term can be written as $\lambda\overline{x}_m.\, a\, \overline{t}_n$, where a is not an application; we call a the *head* of the term. By convention, a, b, and ζ denote heads. If a is a free variable, we call the term *flex*; otherwise, the term is *rigid*.

Deviating from the standard notion of higher-order subterm, we define subterms on β-reduced terms as follows: a term t is a subterm of t at position ε. If s is a subterm of u_i at position p, then s is a subterm of $a\, \overline{u}_n$ at position $i.p$. If s is a subterm of t at position p, then s is a subterm of $\lambda x.\, t$ at position $1.p$. Our definition of subterm gracefully generalizes the corresponding first-order notion: a is a subterm of $\mathsf{f\,a\,b}$ at position 1, but f and $\mathsf{f\,a}$ are not subterms of $\mathsf{f\,a\,b}$. We say a term is ground if it has no free variables, and closed if it has no loose bound variables.

Throughout this thesis, we consider completeness of higher-order calculi only with respect to Henkin semantics [20]. Note that no complete, consistent calculus can exist for higher-order calculi with standard (full) semantics.

Some of the approaches described in this thesis are based on *rank-1 polymorphism* [31, 88]. This is an extension of simply typed logic which adds type variables to the definition of types and type arguments to polymorphic constants (Chapter 5). It further requires that type arguments are instantiated with concrete, non-quantified types. In a clausal structure it allows (implicit) quantification over types only on top-level and not inside literals.

2.4 Clausal Forms

The standard resolution and superposition calculi do not work directly on formulas, but on normal forms, *clauses*. Thus, the initial problem, expressed as a set of formulas, must be transformed into a set of clauses. For propositional and first-order logic, there exists such a transformation that does not affect satisfiability of the problem [126]. In higher-order logic, formulas are first-class citizens and higher-order calculi can in general work with the nonnormalized problem, in its original form. However, resolution and superposition-based higher-order calculi still perform best when the problem is clausified using a transformation similar to the one used in the first-order case. We have analyzed different approaches to clausification in a higher-order context [17].

An *equation* $s \approx t$ corresponds to an unordered pair of terms. A *literal* l is an equation $s \approx t$ or its negation (disequation) $s \not\approx t$. A clause C is a finite multiset of literals, interpreted and written disjunctively: $l_1 \vee \cdots \vee l_n$. Free variables of clauses are implicitly universally quantified. Note that clause-level operators ($\vee, \approx, \not\approx$) are not typeset in bold to distinguish them from term-building symbols ($\mathbf{\vee}, \mathbf{\approx}, \mathbf{\not\approx}, \mathbf{\wedge}, \mathbf{\forall}, \ldots$) which are typeset in bold. In standard, non-clausal first-order logic, an atom is either predicate or equational. For uniformity, and to stay close to the implementation, we encode predicate atoms as equations with T. Negative literals are encoded as disequations. For example, $\mathsf{even}(x)$ is encoded as $\mathsf{even}(x) \approx \mathsf{T}$, and $\neg\, \mathsf{even}(x)$ is encoded as $\mathsf{even}(x) \not\approx \mathsf{T}$. To lighten the notation, we sometimes write the predicate literals in nonencoded form. Applying a substitution to a literal is reduced to applying it to both sides of the (dis)equation and it is extended pointwise to clauses. We say $\sigma(C)$ is a ground instance of C if $\sigma(C)$ has no free variables.

2.5 Superposition

Superposition is one of the most successful calculi for first-order logic with equality. It owes its success to a careful treatment of equality and built-in heuristics to prune the search space such as term order and selection functions.

This calculus works on problems in clausified form and proves the problem by refuting its negation. In practice this means that, to apply superposition to a given problem, we first must negate its conjecture and clausify the whole problem using some of the clausification algorithms [126, 137]. The calculus then applies its inference rules to the clauses, adding results of the inferences to the working set of clauses. As this process is quite prolific, it also uses an order on terms which gives an indication of a term's "simplicity" to determine if some clauses can be simplified or eliminated from the working set. Superposition is refutationally complete: This means that, if results of all inferences are computed in a systematic, fair manner, the empty clause ($\bot$) will eventually be derived for a provable (i.e., valid) problem. As first-order logic is undecidable, no guarantees are given for unprovable (i.e., invalid) problems. Before introducing the rules of the calculus, we provide definitions of the background concepts.

2.5.1 Term Order and Selection

Superposition calculus is parameterized by a term order $\succ$ with the following properties:

Irreflexive	For all terms s, $s \not\succ s$
Transitive	For all terms s, t, u, if $s \succ t$ and $t \succ u$ then $s \succ u$
Subterm property	For all terms s and contexts C, $C[s] \succ s$
Respects substitutions	For all terms s, t and substitutions σ, $s \succ t$ implies $\sigma(s) \succ \sigma(t)$
Respects contexts	For all terms s, t and nonempty contexts C, $s \succ t$ implies $C[s] \succ C[t]$
Ground total	For all distinct ground terms s and t either $s \succ t$ or $t \succ s$
Well-founded	There are no infinite chains of the form $s_1 \succ s_2 \succ \cdots$

The term order is lifted to literals and clauses using the *multiset extension* of $\succ$. The order is extended to multisets as follows [8, Sect. 2.5]. For two multisets S_1 and S_2, we write $S_1 \succ S_2$ if $S_1 \neq S_2$ and whenever $S_2(x) > S_1(x)$ then there is some $y \succ x$ such that $S_1(y) > S_2(y)$. We use the notation $S(x)$ to denote the number of occurrences of x in S. To use the multiset extension, positive literals are represented as $\{\{s\},\{t\}\}$ and negative ones as $\{\{s, t\}\}$, which makes the negative literals slightly greater than the positive ones. Clauses are then represented as multisets of such literals.

Two orders that are commonly used in superposition theorem proving are the Knuth–Bendix order (KBO) [5, Sect. 5.4.4] and the lexicographic path order (LPO) [5, Sect. 5.4.2]. KBO assigns integer weights to symbols and uses precedence between symbols to break eventual ties. LPO entirely relies on precedence and inspects term structures more closely to determine the order.

The *literal selection function* selects a (multi)subset of literals from a given (multi)set of literals. Superposition requires that at least one of the selected literals is negative.

2.5.2 Unification

Calculi from the resolution family, including superposition, perform inferences only on unifiable terms. We say terms s and t are *unifiable* if there is a substitution σ such that $\sigma(s) = \sigma(t)$; we further say σ is a unifier. A *unification constraint* $s \stackrel{?}{=} t$ is a pair of two terms of the same type. We always specify if the constraint should be interpreted as ordered or unordered. We say a substitution σ is *more general* than ϱ if there exists a substitution θ such that $\varrho = \theta\sigma$. A *most general unifier* (MGU) is a unifier σ, such that for any other unifier ϱ, σ is more general than ϱ. In first-order logic, the MGU is unique up to variable renaming. For example, $\mathsf{f}(X) \stackrel{?}{=} \mathsf{f}(Y)$ yields the MGU $\{X \mapsto Y\}$ or $\{Y \mapsto X\}$. First-order logic also admits efficient, linear-time algorithms for computing MGUs [79].

In higher-order logic, unification is performed modulo $\alpha\beta\eta$-equivalence. Under these conditions, the uniqueness of the MGU is not guaranteed. Consider the constraint $X(\mathsf{fa}) \stackrel{?}{=} \mathsf{f}(X\,\mathsf{a})$. Any substitution of the form $\{X \mapsto \lambda x.\, \mathsf{f}^i\, x\}$ is a unifier, with $\mathsf{f}^i\, x$ denoting iterated, i-fold application of f. However, neither of these substitutions is more general than the other. To generalize MGUs to higher-order logic, the concept of the *complete sets of unifiers* (CSU) was introduced.

A (higher-order) unifier of a multiset of unification constraints E is a substitution σ, such that $\sigma(s) \longleftrightarrow^*_{\alpha\beta\eta} \sigma(t)$, for all $s \stackrel{?}{=} t \in E$. A complete set of unifiers of E is defined as a set U of E's unifiers along with a set V of *auxiliary variables* such that no $s \stackrel{?}{=} t \in E$ contains variables from V and for every unifier ρ of E there exists a $\sigma \in U$ and a substitution θ such

that for all $X \notin V$, $\rho(X) = \theta(\sigma(X))$. The MGU, when it exists, corresponds to a one-element CSU. A unifier of terms s and t is a unifier of the singleton multiset $\{s \stackrel{?}{=} t\}$. There is no algorithm to decide if two higher-order terms are unifiable, but there exist procedures that enumerate all elements of the CSU for two terms.

These higher-order concepts gracefully generalize the corresponding first-order ones. For example, if s at t are first-order, then there must exist a singleton CSU, i.e., the MGU.

2.5.3 Inference Rules

Before spelling out the inference rules of standard, first-order superposition, let us introduce some helpful notation, following Schulz [143]. With $t|_p$ we denote the subterm of t at position p and with $t[p \longleftarrow t']$ we denote the term obtained by replacing this subterm with t'. Let *Sel* be a selection function, $\succ$ a term order, $C = l \vee C'$ a clause, and σ a substitution. We say that $\sigma(l)$ is eligible for resolution if: (1) nothing is selected by *Sel* and $\sigma(l)$ is $\succ$-maximal within the literals in $\sigma(C)$ or (2) *Sel* selected some literals and $\sigma(l)$ is the maximal within either positive or negative selected literals. We say $\sigma(l)$ is eligible for paramodulation if it is positive, nothing is selected, and $\sigma(l)$ is the maximal literal within the literals of $\sigma(C)$. With $mgu(s,t)$ we denote the MGU of s and t. Using a horizontal line to separate premises and the conclusion, the four inference rules of superposition are as follows:

Equality resolution (ER)

$$\frac{s \not\approx t \vee C}{\sigma(C)}\,\mathrm{ER}$$

where $\sigma = mgu(s,t)$, and $\sigma(s \not\approx t)$ is eligible for resolution

Equality factoring (EF)

$$\frac{s \approx t \vee u \approx v \vee C}{\sigma(t \not\approx v \vee u \approx v \vee C)}\,\mathrm{EF}$$

where $\sigma = mgu(s,u), \sigma(t) \not\succ \sigma(s)$ and $\sigma(s \approx t)$ is eligible for paramodulation

Superposition into positive literals (SP)

$$\frac{s \approx t \vee C \qquad u \approx v \vee D}{\sigma(u[p \longleftarrow t] \approx v \vee C \vee D)}\,\mathrm{SP}$$

where $\sigma = mgu(u|_p, s), \sigma(t) \not\succ \sigma(s), \sigma(v) \not\succ \sigma(u)$, $\sigma(s \approx t)$ and $\sigma(u \approx v)$ are eligible for paramodulation, and $u|_p$ is not a variable

Superposition into negative literals (SN)

$$\frac{s \approx t \vee C \qquad u \not\approx v \vee D}{\sigma(u[p \longleftarrow t] \not\approx v \vee C \vee D)}\,\mathrm{SN}$$

where $\sigma = mgu(u|_p, s), \sigma(t) \not\succ \sigma(s), \sigma(v) \not\succ \sigma(u)$, $\sigma(s \approx t)$ is eligible for paramodulation, $\sigma(u \not\approx v)$ is eligible for resolution, and $u|_p$ is not a variable

We call these four rules *generating rules* as their conclusions are added to the set of clauses. In the rest of this thesis we denote generating rules with a horizontal bar separating premises and conclusions.

Let us give some intuition behind how these rules are applied in practice. The ER rule is used to establish reflexivity of equality—the property that each term is equal to itself. Consider the formula $\forall x.\, x \approx 2 \longrightarrow \mathsf{even}(x)$. In clausal form it is $X \not\approx 2 \vee \mathsf{even}(X)$. Applying the ER rule effectively applies the precondition $X \approx 2$ to the conclusion $\mathsf{even}(X)$ to obtain $\mathsf{even}(2)$.

Similarly to the factoring rule in the standard resolution calculus, the EF rule can be used to find instances of the clause that duplicate some literals. Additionally, it takes care of some rarely occurring edge cases that are a consequence of restrictive side-conditions of other rules of the calculus. Consider the clause $\mathsf{f}(X) \approx \mathsf{a} \vee \mathsf{f}(Y) \approx \mathsf{a} \vee \mathsf{p}(X, Y)$. Applying EF to its first two literals yields $\mathsf{a} \not\approx \mathsf{a} \vee \mathsf{f}(X) \approx \mathsf{a} \vee \mathsf{p}(X, X)$ (with $\sigma = \{Y \mapsto X\}$). This clause then further simplifies to $\mathsf{f}(X) \approx \mathsf{a} \vee \mathsf{p}(X, X)$ using ER.

Lastly, the rules SP and SN are used to simulate what mathematicians do when they replace equals by equals. In calculi preceding superposition, the name paramodulation is often used for rules that replace equals by equals. Note that with *the superposition rule* we will refer to both SN and SP. The left premise intuitively states that an equation $s \approx t$ holds under condition C. Similarly, the right premise asserts that a (dis)equation holds under a condition. The superposition rule simply concatenates the conditions for both premises and replaces equals by equals in two main inference terms. For example, given clauses $\neg\mathsf{even}(X) \vee \mathsf{f}(X) \approx \mathsf{a}$ and $\neg\mathsf{even}(\mathsf{b}) \vee \mathsf{g}(\mathsf{f}(Y)) \approx \mathsf{b}$, superposition between their last literals results in $\neg\mathsf{even}(Y) \vee \neg\mathsf{even}(\mathsf{b}) \vee \mathsf{g}(\mathsf{a}) \approx \mathsf{b}$.

2.5.4 The Redundancy Criterion and Simplification Rules

The rules described above are used to generate new clauses from already derived ones. Even though orders and selection are used to reduce the search space, many unnecessary clauses might be created. To identify unnecessary clauses in the search space, superposition features a *redundancy criterion* [8, Sect. 4.2.2]. We say that a ground clause C is *redundant* in a ground set of clauses N if it is entailed by a subset of clauses in N such that each clause in this subset is $\succ$-smaller than C. More generally, a clause (ground or not) is redundant in set N if every ground instance of C is redundant for the grounding of N. Removing redundant clauses does not affect completeness of superposition.

Next to the four generating rules, superposition provers also use simplification rules, which are justified by the redundancy criterion. These rules replace the premises with conclusions, and we denote them using double bars. In theory, the rules work by adding conclusions and then using them to show that premises are redundant; after this, premises can be eliminated. As an example of such a rule, let us introduce rewriting (demodulation) of negative literals (RN):

$$\frac{s \approx t \qquad u \not\approx v \vee D}{\;\;s \approx t \qquad u[p \leftarrow \sigma(t)] \not\approx v \vee D\;\;}\,\mathrm{RN}$$

where $\sigma(s) = u|_p$ and $\sigma(s) \succ \sigma(t)$.

Simple rules such as removing duplicate literals, literals of the form $s \not\approx s$, or tautological clauses are clearly justified by the redundancy criterion. Schulz gives an extensive list of such rules [143]. This list includes the rule which allows rewriting positive literals but has more side conditions than RN.

We say that a clause C subsumes clause D if there is a substitution σ such that $\sigma(C) \subseteq D$. Subsumption is one of the most important operations of a theorem prover. This rule does not adhere to the redundancy criterion, but can be justified using other mechanisms [171].

2.5.5 The Saturation Procedure

Superposition provers saturate the input problem with respect to the calculus's inference rules using the *given clause procedure* [4, 116]. It partitions the proof state into a passive set $\mathscr{P}$ and an active set $\mathscr{A}$. All clauses start in $\mathscr{P}$. At each iteration of the procedure's main loop, the prover chooses a clause C from $\mathscr{P}$, simplifies it, and moves it to $\mathscr{A}$ (i.e., it *activates* it). Then all inferences between C and active clauses are performed. The resulting clauses are again simplified and put in $\mathscr{P}$. We call the pair $(\mathscr{P}, \mathscr{A})$ the *proof state*. Provers differ in which clauses are used for simplification: Otter-loop [116] provers use both active and passive clauses whereas DISCOUNT-loop [4] provers use only active clauses. The provers we discuss in this thesis, E and Zipperposition, are both DISCOUNT-loop provers.

2.5.6 Higher-Order Superposition Calculi

In Chapter 1 we described a roadmap (consisting of three steps) to extend an efficient superposition prover to higher-order logic. My colleagues and me did the necessary theoretical work by extending the superposition calculus using the same roadmap:

1. Bentkamp, Blanchette, Cruanes, and Waldmann designed a complete superposition calculus for higher-order logic devoid of λ-abstraction and first-class Booleans [15]. This calculus is called the *Boolean-free λ-free higher-order superposition calculus* and we shortly refer to it as λfSup.

2. Bentkamp, Blanchette, Tourret, Vukmirović, and Waldmann extended this calculus to support λ-abstraction [18]. They call this new calculus the *Boolean-free λ-superposition calculus*, and we shortly refer to it as λSup.

3. Bentkamp, Blanchette, Tourret, and Vukmirović extended λSup with support for first-class Booleans [17]. This amounts to designing the complete calculus for full higher-order logic. This calculus is called Boolean λ-superposition, or shortly $o\lambda$Sup.

Note that in this thesis the name λ-superposition is used for both λSup and $o\lambda$Sup.

2.5.7 Theorem Provers

All the techniques described in this thesis have been implemented in two superposition theorem provers: Zipperposition [48, 49] and E [143]. The former was used as a testbed for prototyping ideas, while the latter is the main target of our work as it is more efficient.

Zipperposition Zipperposition is a higher-order theorem prover implementing λfSup, λSup, $o\lambda$Sup, and other superposition-like calculi (including Bhayat and Reger's combinatory superposition [25]). The prover was conceived as a testbed for rapidly experimenting with extensions of first-order superposition, but over time it has assimilated many of E's techniques and heuristics and become quite powerful. Still, its first-order performance is a far cry from E's. Zipperposition is written in OCaml, and features a modular system for adding new rules and techniques to implemented calculi. This makes it attractive for fast evaluation of various promising extensions of superposition.

E E is a state-of-the-art first-order prover based on superposition. In the last decade, together with its derivatives, it was usually the second place at the first-order division of the CASC theorem prover competition [163]. It is open-source, written in C (without using any external libraries), and designed around efficient algorithms and data structures rather than a highly optimized codebase. This makes it a good target for implementing successful higher-order techniques previously evaluated in Zipperposition.

3

Extending a Brainiac Prover to Lambda-Free Higher-Order Logic

Joint work with
Jasmin Blanchette, Simon Cruanes and Stephan Schulz

Decades of work have gone into developing efficient proof calculi, data structures, algorithms, and heuristics for first-order automatic theorem proving. Higher-order provers lag behind in terms of efficiency. Instead of developing a new higher-order prover from the ground up, we propose to start with the state-of-the-art superposition prover E and gradually enrich it with higher-order features. We explain how to extend the prover's data structures, algorithms, and heuristics to λ-free higher-order logic, a formalism that supports partial application and applied variables. Our extension outperforms the traditional encoding and forms a stepping stone toward full higher-order logic.

In this work I designed, implemented and evaluated all changes to term representation, algorithms and indexing data structures. Jasmin Blanchette came up with the extension of fingerprinting indexing. Stephan Schulz provided the necessary E expertise.

3.1 Introduction

Superposition provers such as E [147], SPASS [173], and Vampire [100] are among the most successful first-order reasoning systems. They serve as backends in various frameworks, including software verifiers (e.g., Why3 [64]), automatic higher-order theorem provers (e.g., Leo-III [153], Satallax [39]), and one-click "hammers" in proof assistants (e.g., HOLyHammer in HOL Light [89], Sledgehammer in Isabelle [131]).

Decades of research have gone into refining calculi, devising efficient data structures and algorithms, and developing heuristics to guide proof search [146]. This work has mostly focused on first-order logic with equality.

Research on higher-order automatic provers has resulted in systems such as LEO [19], Leo-II [22], and Leo-III [153], based on resolution and paramodulation, and Satallax [39], based on tableaux and SAT solving. They feature a "cooperative" architecture, pioneered by LEO: They are full-fledged higher-order provers that regularly invoke an external first-order prover with a low time limit as a terminal procedure, in an attempt to finish the proof quickly using only first-order reasoning. However, the first-order backend will succeed only if all the necessary higher-order reasoning has been performed, meaning that much of the first-order reasoning is carried out by the slower higher-order prover. As a result, this architecture leads to suboptimal performance on largely first-order problems, such as those that often arise in interactive verification [156]. For example, at the 2017 installment of the CADE ATP System Competition (CASC) [159], Leo-III, which uses E as a backend, proved 652 out of 2000 first-order problems in the Sledgehammer division, compared with 1185 for E on its own and 1433 for Vampire.

To obtain better performance, we propose to start with a competitive first-order prover and extend it to full higher-order logic one feature at a time. Our goal is a *graceful* extension, so that the system behaves as before on first-order problems, performs mostly like a first-order prover on typical, mildly higher-order problems, and scales up to arbitrary higher-order problems, in keeping with the zero-overhead principle: *What you don't use, you don't pay for.*

As a stepping stone toward full higher-order logic, we initially restrict our focus to a higher-order logic without λ-expressions (Sect. 3.2). Compared with first-order logic, its distinguishing features are partial application and applied variables. It is rich enough to express the recursive equations of higher-order combinators, such as map on lists:

$$\mathsf{map}\ f\ \mathsf{nil} \approx \mathsf{nil} \qquad\qquad \mathsf{map}\ f\ (\mathsf{cons}\ x\ xs) \approx \mathsf{cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

Our vehicle is E [143, 147], a prover developed primarily by Schulz. It is written in C and offers good performance. It gets its "brainiac" name by emphasizing intelligent heuristics more than raw speed. E regularly scores among the top systems at CASC and is usually the strongest open-source prover in the relevant divisions. It also serves as a backend for competitive higher-order provers. We refer to our extended version of E as Ehoh. It corresponds to a prerelease version of E 2.5 configured with the option `--enable-ho`.[1]

The main challenges we faced involved the representation of types and terms (Sect. 3.3), the unification and matching algorithms (Sect. 3.4), and the indexing data structures (Sect.

[1] `https://github.com/eprover/eprover/commit/80946ac`

3.5). We also adapted the inference rules (Sect. 3.6), the heuristics (Sect. 3.7), and the preprocessor (Sect. 3.8).

A central aspect of our work is a set of techniques we call *prefix optimization*. Taking a traditional look at higher-order terms, they contain twice as many proper subterms as first-order terms; for example, $\mathsf{f}\ (\mathsf{g}\ \mathsf{a})\ \mathsf{b}$ contains not only the "argument" subterms $\mathsf{g}\ \mathsf{a}$, a, b but also the "prefix" subterms f, $\mathsf{f}\ (\mathsf{g}\ \mathsf{a})$, g. Many operations, including superposition and rewriting, require traversing all subterms of a term. The prefix optimization allows the prover to traverse subterms recursively in a first-order fashion, and simultaneously consider the prefixes of a given subterm at almost no additional cost. Our experiments (Sect. 3.9) show that Ehoh is almost as fast as E on first-order problems and can also prove higher-order problems that do not require synthesizing λ-terms. In Chapter 7 we show how to add support for λ-terms and higher-order unification.

3.2 Logic

Our logic is a variant of the intensional λ-free Boolean-free higher-order logic (λffHOL) described by Bentkamp et al. [15, Sect. 2], which could also be called "applicative first-order logic." In the spirit of FOOL [98], we extend the syntax of this logic by making formulas a special case of terms (without adding logical symbols to the signature), and its semantics by interpreting the Boolean type o as a two-element domain. Functional extensionality, the property that two functions are equal if they always return the same value when given the same arguments, can be obtained by adding suitable axioms [15, Sect. 3.1].

This logic differs from the higher-order logic described in Sect. 2.3 in three ways. First, λ-abstraction is disallowed. Second, logical connectives are not part of the set of symbols. Instead, there is a special inductive case in the definition of terms that defines formulas. Third, subterms are defined in a more traditional way, as defined below.

For reference, we provide the definition of terms. Terms, ranged over by s, t, u, v, are either *variables* $x, y, z, \ldots$, (*function*) *symbols* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{f}, \mathsf{g}, \ldots$ (often called "constants" in the higher-order literature), binary applications $s\ t$, or Boolean terms $\top$, $\bot$, $\neg\, s$, $s \wedge t$, $s \vee t$, $s \longrightarrow t$, $s \longleftrightarrow t$, $\forall x.\, s$, $\exists x.\, s$, $s \approx t$. E and Ehoh clausify the input as a preprocessing step, producing a clause set in which the only proper Boolean subterms are variables, $\top$, and $\bot$. Note that we use lowercase letters for free variables as bound variables do not appear in clauses. A term's *arity* is the number of extra arguments it can take. If ι is a base type, f has type $\iota \longrightarrow \iota \longrightarrow \iota$, and a has type ι, then f is binary, $\mathsf{f}\ \mathsf{a}$ is unary, and $\mathsf{f}\ \mathsf{a}\ \mathsf{a}$ is nullary. Subterms are defined in the traditional higher-order way; for example, $s\ t$ has all subterms of s and t as subterms, in addition to $s\ t$ itself; as a consequence $\mathsf{f}\ \mathsf{a}$ is a subterm of $\mathsf{f}\ \mathsf{a}\ \mathsf{a}$. With $\mathcal{V}ar(x)$ we denote the set of free variables of x where x ranges over terms, clauses, set of clauses, or any other objects that contain terms.

In this chapter, substitutions σ are partial functions of finite domain from variables to terms, written $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, where each s_i has the same type as x_i. The substitution $\sigma[x \mapsto s]$ maps x to s and otherwise coincides with σ. Applying σ to a variable beyond σ's domain is the identity. We deviated from the view of substitutions in Sect. 2.3 as it made proofs in Sect. 3.4 and 3.5 easier. It is easy to check that both views are equivalent. We also consider unification constraints $s \stackrel{?}{=} t$ as *ordered* pairs.

A well-known technique to support λffHOL is to use the *applicative encoding*: Every n-ary symbol is mapped to a nullary symbol, and application is represented by a distin-

guished binary symbol @. Thus, the λfHOL term $\mathtt{f}\ (x\ \mathtt{a})\ \mathtt{b}$ is encoded as the first-order term $@(@(\mathtt{f}, @(x, \mathtt{a})), \mathtt{b})$. However, this representation is not graceful, since it also introduces @'s for terms within λfHOL's first-order fragment. By doubling the size and depth of terms, the encoding clutters data structures and slows down term traversals. In our empirical evaluation, we find that the applicative encoding can decrease the success rate by up to 15% (Sect. 3.9). For these and further reasons, it is not ideal (Sect. 3.10).

3.3 Types and Terms

The term representation is a central concern when building a theorem prover. Delicate changes to E's representation were needed to support partial application and especially applied variables. In contrast, the introduction of a higher-order type system had a less dramatic impact on the prover's code.

Types For most of its history, E supported only untyped first-order logic. Cruanes implemented support for atomic types for E 2.0 [48, p. 117]. Symbols are declared with a type signature: $\mathtt{f} : (\tau_1, \ldots, \tau_n) \to \tau$. Atomic types are represented by integers, leading to efficient type comparisons.

In λfHOL, function types are built using the type constructor →, which can be arbitrarily nested—e.g., $(\iota \to \iota) \to \iota$. A natural way to represent such types is to mimic their recursive structure using a tagged union. However, this leads to memory fragmentation; a simple operation such as querying the type of a function's ith argument would require dereferencing i pointers. We prefer a flattened representation, in which a type $\tau_1 \to \cdots \to \tau_n \to \iota$ is represented by a single node labeled with → and pointing to the array $(\tau_1, \ldots, \tau_n, \iota)$.

Ehoh stores all types in a shared bank and implements perfect sharing, ensuring that types that are structurally the same are represented by the same object in memory. Type equality can then be implemented as a pointer comparison.

Terms In E, terms are stored as perfectly shared directed acyclic graphs [109]. Each node, or *cell*, contains 11 fields, including `f_code`, an integer that identifies the term's head symbol (if ≥ 0) or variable (if < 0); `arity`, an integer corresponding to the number of arguments passed to the head; `args`, an array of size `arity` consisting of pointers to arguments; and `binding`, which may store a substitution for a variable (if `f_code` < 0), used for unification and matching.

In first-order logic, the arity of a variable is always 0, and the arity of a symbol is given by its type signature. In higher-order logic, variables may have function type and be applied, and symbols can be applied to fewer arguments than specified by their type signatures. A natural representation of λfHOL terms as tagged unions would distinguish between variables x, symbols f, and binary applications $s\ t$. However, this scheme suffers from memory fragmentation and linear-time access, as with the representation of types, affecting performance on purely or mostly first-order problems. Instead, we propose a flattened representation, as a generalization of E's existing data structures: Allow arguments to variables, for symbols let `arity` be the number of actual arguments, and rename the field to `num_args`. This representation, often called "spine notation," is isomorphic to the stan-

dard definition of higher-order terms with binary application. It is employed in various higher-order reasoning systems, including Leo-III [153] and Zipperposition [48, 49].

A side effect of the flattened representation is that prefix subterms are not shared. For example, the terms f a and f a b correspond to the flattened cells f(a) and f(a, b). The argument subterm a is shared, but not the prefix f a. Similarly, x and x b are represented by two distinct cells, $x()$ and $x(\mathsf{b})$, and there is no connection between the two occurrences of x. In particular, when $x()$'s `binding` field is updated, this does not affect the `binding` of $x(\mathsf{b})$.

A potential solution would be to systematically traverse a clause and set the `binding` fields of all cells of the form $x(\overline{s})$ whenever a variable x is bound, but this would be inefficient and inelegant. Instead, we implemented a hybrid approach: Variables are applied by an explicit application operator @, to ensure that they are always perfectly shared. Thus, x b c is represented by the cell $@(x, \mathsf{b}, \mathsf{c})$, where x is a shared subcell. This is graceful, since variables never occur applied in first-order terms. The main drawback is that some normalization is necessary after substitution: Whenever a variable is instantiated by a symbol-headed term, the @ symbol must be eliminated. Applying the substitution $\{x \mapsto \mathsf{f}\ \mathsf{a}\}$ to the cell $@(x, \mathsf{b}, \mathsf{c})$ must produce $\mathsf{f}(\mathsf{a}, \mathsf{b}, \mathsf{c})$ and not $@(\mathsf{f}(\mathsf{a}), \mathsf{b}, \mathsf{c})$, for consistency with other occurrences of f a b c.

There is one more complication related to the `binding` field. In E, it is easy and useful to traverse a term as if a substitution has been applied, by following all set `binding` fields. In Ehoh, this is not enough, because cells must also be normalized. To avoid repeatedly creating the same normalized cells, we introduced a `binding_cache` field that connects a $@(x, \overline{s})$ cell with its substitution. However, this cache can easily become stale when x's `binding` pointer is updated. To detect this situation, we store x's `binding` value in the $@(x, \overline{s})$ cell's `binding` field (which is otherwise unused). To find out whether the cache is valid, it suffices to check that the `binding` fields of x and $@(x, \overline{s})$ are equal.

Term Orders Superposition provers rely on term orders to prune the search space. The order must be a reduction order (Sect. 2.5.1). E implements both the Knuth–Bendix order (KBO) and the lexicographic path order (LPO). KBO is widely regarded as the more robust option for superposition. In earlier work, Blanchette and colleagues have shown that only KBO can be generalized gracefully while preserving the necessary properties for superposition [13, 32]. For this reason, in this chapter, we focus on KBO.

E implements Löchner's linear-time algorithm for KBO [108], which relies on the tupling method to store intermediate results. It is straightforward to generalize the algorithm to compute the graceful λfHOL version of KBO [13]. The main difference is that when comparing two terms $\mathsf{f}\ \overline{s}_m$ and $\mathsf{f}\ \overline{t}_n$, because of partial application it is possible that $m \neq n$; this required changing the implementation to perform a length-lexicographic comparison of the tuples $\overline{s}_m$ and $\overline{t}_n$.

Input and Output Syntax E implements the TPTP [157] formats FOF and TF0, corresponding to untyped and monomorphic first-order logic, for both input and output. In Ehoh, we added support for the λfHOL fragment of TPTP TH0, which provides monomorphic higher-order logic. Thanks to the use of a standard format, Ehoh's proofs can immediately be parsed by Sledgehammer [131], which reconstructs them using a variety of techniques. There is ongoing work on increasing the level of detail of E's proofs, to

facilitate proof interchange and independent proof checking [135]; this will also benefit Ehoh.

3.4 Unification and Matching

Syntactic unification of λfHOL terms has a first-order flavor. It is decidable, and most general unifiers (MGUs) are unique up to variable renaming. For example, the unification constraint $\mathsf{f}\ (x\ \mathsf{a}) \stackrel{?}{=} x\ (\mathsf{f}\ \mathsf{a})$, used to illustrate an infinite set of independent unifiers in full higher-order logic (Sect. 2.5.2), has the MGU $\{x \mapsto \mathsf{f}\}$ in λfHOL. Matching is a special case of unification where only the variables on the left-hand side can be instantiated.

An easy but inefficient way to implement unification and matching for λfHOL is to apply the applicative encoding (Sect. 3.2), perform first-order unification or matching, and decode the resulting substitution. To avoid overhead, we generalize the first-order unification and matching procedures to operate directly on λfHOL terms.

3.4.1 Unification

We present our unification procedure as a nondeterministic transition system that generalizes the one of Baader and Nipkow [5]. A unification problem consists of a finite set S of unification constraints $s_i \stackrel{?}{=} t_i$, where s_i and t_i are of the same type. A problem is in *solved form* if it has the form $\{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$, where the x_i's are distinct and do not occur in the t_j's. The corresponding unifier is $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. The transition rules attempt to bring the input constraints into solved form. They can be applied in any order and eventually reach a normal form, which is either an idempotent MGU expressed in solved form or the special value $\bot$, denoting unsatisfiability of the constraints.

The first group of rules—the *positive* rules—consists of operations that focus on a single constraint and replace it with a new (possibly empty) set of constraints:

Delete	$\{t \stackrel{?}{=} t\} \uplus S \Longrightarrow S$
Decompose	$\{\mathsf{f}\ \overline{s}_m \stackrel{?}{=} \mathsf{f}\ \overline{t}_m\} \uplus S \Longrightarrow S \cup \{s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\}$
DecomposeX	$\{x\ \overline{s}_m \stackrel{?}{=} u\ \overline{t}_m\} \uplus S \Longrightarrow S \cup \{x \stackrel{?}{=} u, s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\}$ if x and u have the same type and $m > 0$
Orient	$\{\mathsf{f}\ \overline{s} \stackrel{?}{=} x\ \overline{t}\} \uplus S \Longrightarrow S \cup \{x\ \overline{t} \stackrel{?}{=} \mathsf{f}\ \overline{s}\}$
OrientXY	$\{x\ \overline{s}_m \stackrel{?}{=} y\ \overline{t}_n\} \uplus S \Longrightarrow S \cup \{y\ \overline{t}_n \stackrel{?}{=} x\ \overline{s}_m\}$ if $m > n$
Eliminate	$\{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \{x \stackrel{?}{=} t\} \cup \{x \mapsto t\}(S)$ if $x \in \mathcal{V}ar(S) \setminus \mathcal{V}ar(t)$

The Delete, Decompose, and Eliminate rules are essentially as for first-order terms. The Orient rule is generalized to allow applied variables and complemented by a new OrientXY rule. DecomposeX, also a new rule, can be seen as a variant of Decompose that analyzes applied variables; the term u may be an application.

The rules of the second group—the *negative* rules—detect unsolvable constraints:

Clash	$\{\mathsf{f}\ \overline{s} \stackrel{?}{=} \mathsf{g}\ \overline{t}\} \uplus S \Longrightarrow \bot$; if $\mathsf{f} \neq \mathsf{g}$
ClashTypeX	$\{x\ \overline{s}_m \stackrel{?}{=} u\ \overline{t}_m\} \uplus S \Longrightarrow \bot$; if x and u have different types
ClashLenXF	$\{x\ \overline{s}_m \stackrel{?}{=} \mathsf{f}\ \overline{t}_n\} \uplus S \Longrightarrow \bot$; if $m > n$
OccursCheck	$\{x \stackrel{?}{=} t\} \uplus S \Longrightarrow \bot$; if $x \in \mathcal{V}ar(t)$ and $x \neq t$

Clash and OccursCheck are essentially as in Baader and Nipkow. ClashTypeX and ClashLenXF are variants of Clash for applied variables.

The derivation below demonstrates the computation of MGUs for the unification problem $\{x\,(z\,\mathsf{b}\,\mathsf{c}) \stackrel{?}{=} \mathsf{g}\,\mathsf{a}\,(y\,\mathsf{c})\}$:

$$\begin{aligned}
& \{x\,(z\,\mathsf{b}\,\mathsf{c}) \stackrel{?}{=} \mathsf{g}\,\mathsf{a}\,(y\,\mathsf{c})\} \\
\Longrightarrow_{\mathsf{DecomposeX}}\ & \{x \stackrel{?}{=} \mathsf{g}\,\mathsf{a},\ z\,\mathsf{b}\,\mathsf{c} \stackrel{?}{=} y\,\mathsf{c}\} \\
\Longrightarrow_{\mathsf{OrientXY}}\ & \{x \stackrel{?}{=} \mathsf{g}\,\mathsf{a},\ y\,\mathsf{c} \stackrel{?}{=} z\,\mathsf{b}\,\mathsf{c}\} \\
\Longrightarrow_{\mathsf{DecomposeX}}\ & \{x \stackrel{?}{=} \mathsf{g}\,\mathsf{a},\ y \stackrel{?}{=} z\,\mathsf{b},\ \mathsf{c} \stackrel{?}{=} \mathsf{c}\} \\
\Longrightarrow_{\mathsf{Delete}}\ & \{x \stackrel{?}{=} \mathsf{g}\,\mathsf{a},\ y \stackrel{?}{=} z\,\mathsf{b}\}
\end{aligned}$$

E stores open constraints in a double-ended queue. Constraints are processed from the front. New constraints are added at the front if they involve complex terms that can be dealt with swiftly by Decompose or Clash, or to the back if one side is a variable. This delays instantiation of variables and allows E to detect structural clashes early.

During proof search, E repeatedly needs to test a term s for unifiability not only with some other term t but also with t's subterms. Prefix optimization speeds up this test: The subterms of t are traversed in a first-order fashion; for each such subterm $\zeta\ \overline{t}_n$, at most one prefix $\zeta\ \overline{t}_k$, with $k \leq n$, is possibly unifiable with s, by virtue of their having the same arity. For first-order problems, we can only have $k = n$, since all functions are fully applied. Using this technique, Ehoh is virtually as efficient as E on first-order terms.

The transition system introduced above always terminates with a correct answer. Our proofs follow the lines of Baader and Nipkow. The metavariable $\mathcal{R}$ is used to range over constraint sets S and the special value $\bot$. The set of all unifiers of S is denoted by $\mathcal{U}(S)$. Note that $\mathcal{U}(S \cup S') = \mathcal{U}(S) \cap \mathcal{U}(S')$. We let $\mathcal{U}(\bot) = \emptyset$. The notation $S \Longrightarrow^{!} S'$ indicates that $S \Longrightarrow^{*} S'$ and S' is a normal form (i.e., there exists no S'' such that $S' \Longrightarrow S''$). A variable x is *solved* in S if it occurs exactly once in S, in a constraint of the form $x \stackrel{?}{=} t$.

Lemma 3.1. *If $S \Longrightarrow \mathcal{R}$, then $\mathcal{U}(S) = \mathcal{U}(\mathcal{R})$.*

Proof. The rules Delete, Decompose, Orient, and Eliminate are proved as in Baader and Nipkow. OrientXY trivially preserves unifiers. For DecomposeX, the core of the argument is as follows:

$$\begin{aligned}
& \sigma \in \mathcal{U}(\{x\ \overline{s}_m \stackrel{?}{=} u\ \overline{t}_m\}) \\
\text{iff}\quad & \sigma(x\ \overline{s}_m) = \sigma(u\ \overline{t}_m) \\
\text{iff}\quad & \sigma(x)\ \sigma(s_1) \ldots \sigma(s_m) = \sigma(u)\ \sigma(t_1) \ldots \sigma(t_m) \\
\text{iff}\quad & \sigma(x) = \sigma(u),\ \sigma(s_1) = \sigma(t_1),\ \ldots,\ \text{and}\ \sigma(s_m) = \sigma(t_m) \\
\text{iff}\quad & \sigma \in \mathcal{U}(\{x \stackrel{?}{=} u,\ s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\})
\end{aligned}$$

The proof of the problem's unsolvability if rule Clash or OccursCheck is applicable carries over from Baader and Nipkow. For ClashTypeX, the justification is that $\sigma(x\ \overline{s}_m) =$

$\sigma(u\ \overline{t}_m)$ is possible only if $\sigma(x) = \sigma(u)$, which requires x and u to have the same type. Similarly, for ClashLenXF, if $\sigma(x\ \overline{s}_m) = \sigma(\mathsf{f}\ \overline{t}_n)$ with $m > n$, we must have $\sigma(x\ \overline{s}_{m-n}) = \sigma(x)\ \sigma(s_1)\ \ldots\ \sigma(s_{m-n}) = \mathsf{f}$, which is impossible. □

Lemma 3.2. *If S is a normal form, then S is in solved form.*

Proof. Consider an arbitrary unification constraint $s \stackrel{?}{=} t \in S$. We show that in all but one cases, a rule is applicable, contradicting the hypothesis that S is a normal form. In the remaining case, s is a solved variable in S.

Case $s = x$:

- Subcase $t = x$: Delete is applicable.
- Subcase $t \neq x$ and $x \in \mathcal{V}ar(t)$: OccursCheck is applicable.
- Subcase $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \in \mathcal{V}ar(S \setminus \{s \stackrel{?}{=} t\})$: Eliminate is applicable.
- Subcase $t \neq x$, $x \notin \mathcal{V}ar(t)$, and $x \notin \mathcal{V}ar(S \setminus \{s \stackrel{?}{=} t\})$: The variable x is solved in S.

Case $s = x\ \overline{s}_m$ for $m > 0$:

- Subcase $t = \eta\ \overline{t}_n$ for $n \geq m$: DecomposeX or ClashTypeX is applicable, depending on whether x and $\eta\ \overline{t}_{n-m}$ have the same type.
- Subcase $t = y\ \overline{t}_n$ for $n < m$: OrientXY is applicable.
- Subcase $t = \mathsf{f}\ \overline{t}_n$ for $n < m$: ClashLenXF is applicable.

Case $s = \mathsf{f}\ \overline{s}_m$:

- Subcase $t = x\ \overline{t}_n$: Orient is applicable.
- Subcase $t = \mathsf{f}\ \overline{t}_n$: Due to well-typedness, $m = n$. Decompose is applicable.
- Subcase $t = \mathsf{g}\ \overline{t}_n$: Clash is applicable.

Since each constraint is of the form $x \stackrel{?}{=} t$ where x is solved in S, the problem S is in solved form. □

Lemma 3.3. *If the constraint set S is in solved form, then the associated substitution is an idempotent MGU of S.*

Proof. This lemma corresponds to Lemma 4.6.3 of Baader and Nipkow. Their proof carries over to λfHOL. □

Theorem 3.4 (Partial Correctness)**.** *If $S \Longrightarrow^! \bot$, then S has no solutions. If $S \Longrightarrow^! S'$, then S' is in solved form and the associated substitution is an idempotent MGU of S.*

Proof. The first part follows from Lemma 3.1. The second part follows from Lemma 3.1 and Lemmas 3.2 and 3.3. □

Theorem 3.5 (Termination). *The relation $\Longrightarrow$ is well founded.*

Proof. We define an auxiliary notion of weight: $\mathcal{W}(\zeta\ \overline{s}_m) = m + 1 + \sum_{i=1}^{m} \mathcal{W}(s_i)$. Well-foundedness is proved by exhibiting a measure function from constraint sets to quadruples of natural numbers (n_1, n_2, n_3, n_4), where n_1 is the number of unsolved variables in S; n_2 is the sum of all term weights, $\sum_{s \stackrel{?}{=} t \in S} \mathcal{W}(s) + \mathcal{W}(t)$; n_3 is the number of right-hand sides with variable heads, $|\{s \stackrel{?}{=} x\ \overline{t} \in S\}|$; and n_4 is the number of arguments to left-hand side variable heads, $\sum_{x\,\overline{s}_m \stackrel{?}{=} t \in S} m$.

The following table shows that the application of each positive rule lexicographically decreases the quadruple:

	n_1	n_2	n_3	n_4
Delete	$\geq$	$>$		
Decompose	$\geq$	$>$		
DecomposeX	$\geq$	$>$		
Orient	$\geq$	$=$	$>$	
OrientXY	$\geq$	$=$	$=$	$>$
Eliminate	$>$			

The negative rules, which produce the special value $\bot$, cannot contribute to an infinite $\Longrightarrow$ chain. □

A unification algorithm for λfHOL can be derived from the above transition system, by committing to a strategy for applying the rules. An algorithm which closely follows the Ehoh implementation, abstracting away from complications such as prefix optimization is presented below. We assume a flattened representation of terms; as in Ehoh, each variable stores the term it is bound to in its *binding* field (Sect. 3.3). We also rely on a ApplySubst function, which applies the binding to the top-level variable. The algorithm assumes that the terms to be unified have the same type. The pseudocode is as follows:

```
function SwapNeeded(Term s, Term t) is
  return t.head.isVar()
         ∧ (¬ s.head.isVar() ∨ s.num_args > t.num_args)

function Deref(Term s) is
  while s.head.isVar() ∧ s.head.binding ≠ Null do
    s ← ApplySubst(s, s.head.binding)
  return s

function GobblePrefix(Term x, Term t) is
  res ← Null
  if x.type.args is suffix of t.head.type.args then
    pref_len ← t.head.type.arity − x.type.arity
    if pref_len ≤ t.num_args then
      res ← Term(t.head, t.args[1..pref_len])
  return res
```

```
function Unify(Term s, Term t) is
  constraints ← DoubleEndedQueue()
  constraints.prepend(s)
  constraints.prepend(t)
  while ¬ constraints.isEmpty() do
    t ← Deref(constraints.dequeue())
    s ← Deref(constraints.dequeue())
    if s ≠ t then
      if SwapNeeded(s, t) then
        (t, s) ← (s, t)
      if s.head.isVar() then
        x ← s.head
        prefix ← GobblePrefix(x, t)
        if prefix ≠ Null then
          start_idx ← prefix.num_args + 1
          if x occurs in prefix then
            return False
          else
            x.binding ← prefix
        else
          return False
      else if s.head = t.head then
        start_idx ← 1
      else
        return False
      for i ← start_idx to t.num_args do
        s_arg ← s.args[i − start_idx + 1]
        t_arg ← t.args[i]
        if (s_arg.head.isVar() ∨ t_arg.head.isVar()) then
          constraints.append(t_arg)
          constraints.append(s_arg)
        else
          constraints.prepend(s_arg)
          constraints.prepend(t_arg)
  return True
```

3.4.2 Matching

Given s and t, the matching problem consists of finding a substitution σ such that $\sigma(s) = t$. We then write that "t is an instance of s" or "s generalizes t." We are interested in most general generalizations (MGGs). Matching can be reduced to unification by treating variables in t as nullary symbols [5], but E implements matching separately.

Matching can be specified abstractly as a transition system on matching constraints $s_i \lesssim^? t_i$ consisting of the unification rules Decompose, DecomposeX, Clash, ClashTypeX, ClashLenXF (with $\lesssim^?$ instead of $\stackrel{?}{=}$) and augmented with

Double $\{x \lesssim^? t,\ x \lesssim^? t'\} \uplus S \Longrightarrow \bot$; if $t \neq t'$

ClashLenXY $\{x\ \overline{s}_m \lesssim^? y\ \overline{t}_n\} \uplus S \Longrightarrow \bot$; if $x \neq y$ and $m > n$

ClashFX $\{\mathsf{f}\ \overline{s} \lesssim^? x\ \overline{t}\} \uplus S \Longrightarrow \bot$

The matching relation is sound, complete, and well founded. Interestingly, a Delete rule would be unsound for matching. Consider the problem $\{x \lesssim^? x, x \lesssim^? \mathsf{g}\ x\}$. Applying Delete to the first constraint would yield the solution $\{x \lesssim^? \mathsf{g}\ x\}$, even though the original problem is clearly unsolvable.

3.5 Indexing Data Structures

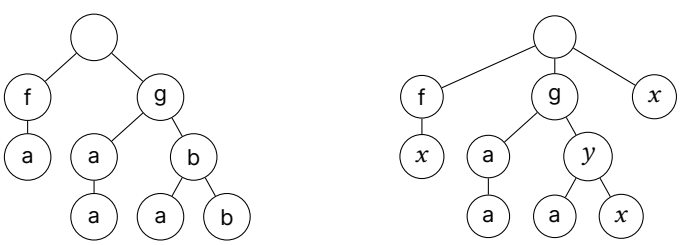
Superposition provers like E work by saturation. Their main loop heuristically selects a clause and searches for potential inference partners among a possibly large set of other clauses. Mechanisms such as simplification and subsumption also require locating terms in a large clause set. For example, when E derives a new equation $s \approx t$, if s is larger than t according to the term order, it will rewrite all instances $\sigma(s)$ of s to $\sigma(t)$ in existing clauses.

To avoid iterating over all terms (including subterms) in large clause sets, superposition provers store the potential inference partners in indexing data structures. A term index stores a set of terms S. Given a *query term* t, a query returns all terms $s \in S$ that satisfy a given *retrieval condition*: $\sigma(s) = \sigma(t)$ (s and t are unifiable), $\sigma(s) = t$ (s generalizes t), or $s = \sigma(t)$ (s is an instance of t), for some substitution σ. *Perfect* indices return exactly the subset of terms satisfying the retrieval condition. In contrast, *imperfect* indices return a superset of eligible terms, and the retrieval condition needs to be checked for each candidate.

E relies on two term indexing data structures, perfect discrimination trees [113] and fingerprint indices [144], that needed to be generalized to λfHOL. It also uses feature vector indices [145] to speed up subsumption and related techniques, but these require no changes to work with λfHOL.

3.5.1 Discrimination Trees

Discrimination trees [113] are tries in which every node is labeled with a symbol or a variable. A path from the root to a leaf corresponds to a "serialized term"—a term expressed without parentheses and commas. Consider the following discrimination trees D_1 and D_2:



Assuming $\mathsf{a}, \mathsf{b}, x, y : \iota$, $\mathsf{f} : \iota \longrightarrow \iota$, and $\mathsf{g} : \iota^2 \longrightarrow \iota$, D_1 represents the term set {f(a), g(a,a), g(b,a), g(b,b)}, and D_2 represents the term set {f(x), g(a,a), g(y,a), g(y,x), x}. E uses perfect discrimination trees for finding generalizations of query terms. Thus, if the query term is g(a,a), it would follow the path g.a.a in D_1 and return {g(a,a)}. For D_2, it would

also explore paths labeled with variables, binding them as it proceeds, and return {g(a, a), g(*y*, a), g(*y*, *x*), *x*}.

It is crucial for this data structure that distinct terms always give rise to distinct serialized terms. Conveniently, this property also holds for λfHOL terms. Suppose that two distinct λfHOL terms yield the same serialization. Clearly, they must disagree on parentheses; one will have the subterm *s t u* where the other has *s* (*t u*). However, these two subterms cannot both be well typed.

When generalizing the data structure to λfHOL, we face a complication due to partial application. First-order terms can only be stored in leaf nodes, but in Ehoh we must also be able to represent partially applied terms, such as f, g, or g a (assuming, as above, that f is unary and g is binary). Conceptually, this can be solved by storing a Boolean on each node indicating whether it is an accepting state (i.e., if it corresponds to a term in the indexed set). In the implementation, the change is more subtle, because several parts of E's code implicitly assume that only leaf nodes are accepting.

The main difficulty specific to λfHOL concerns applied variables. To enumerate all generalizing terms, E needs to backtrack from child to parent nodes. This is achieved using two stacks that store subterms of the query term: T stores the terms that must be matched in turn against the current subtree, and P stores, for each node from the root to the current subtree, the corresponding processed term.

Let $[a_1, \ldots, a_n]$ denote an n-item stack with a_1 on top. Given a query term t, the matching procedure starts at the root with $\sigma = \emptyset$, $T = [t]$, and $P = []$. The procedure advances by repeatedly moving to a suitable child node:

A. If the node is labeled with a symbol f and the top item t of T is of the form $\mathsf{f}(\overline{t}_n)$, replace t by n new items $t_1, \ldots, t_n$, and push t onto P.

B. If the node is labeled with a variable x, there are two subcases. If x is already bound, check that $\sigma(x) = t$; otherwise, extend σ so that $\sigma(x) = t$. Next, pop the term t from T and push it onto P.

The goal is to reach an accepting node. If the query term and all the terms stored in the tree are first-order, T will then be empty, and the entire query term will have been matched. Backtracking works in reverse: Pop a term t from P; if the current node is labeled with an n-ary symbol, discard T's topmost n items; push t onto T. Undo any variable bindings.

As an example, looking up g(b, a) in the tree D_1 would result in the following succession of stack states, starting from the root ε along the path g.b.a:

	ε	g	g.b	g.b.a
σ:	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
T:	[g(b, a)]	[b, a]	[a]	[ ]
P:	[ ]	[g(b, a)]	[b, g(b, a)]	[a, b, g(b, a)]

Backtracking amounts to moving leftward: To get back from g to the root, we pop g(b, a) from P, we discard two items from T, and we push g(b, a) onto T.

To adapt the procedure to λfHOL, the key idea is that an applied variable is not very different from an applied symbol. A node labeled with an n-ary head ζ matches a prefix t' of the k-ary term t popped from T and leaves $n - k$ arguments $\overline{u}$ to be pushed back, with

$t = t'\ \overline{u}$. If ζ is a variable, it must be bound to the prefix t' assuming ζ and t' are of same type. Backtracking works analogously: Given the arity n of the node label ζ and the arity k of the term t popped from P, we discard the topmost $n - k$ items $\overline{u}$ from P.

To illustrate the procedure, we consider the tree D_2 but change y's type to $\iota \to \iota$. This tree stores {f x, g a a, g (y a), g (y x), x}. Let g (g a b) be the query term. We have the following sequence of substitutions σ and stacks T, P:

ε	g	g.y	g.y.x
$\emptyset$	$\emptyset$	$\{y \mapsto \mathsf{g\,a}\}$	$\{y \mapsto \mathsf{g\,a}, x \mapsto \mathsf{b}\}$
[g (g a b)]	[g a b]	[b]	[ ]
[ ]	[g (g a b)]	[g a b, g (g a b)]	[b, g a b, g (g a b)]

When backtracking from g.y to g, by comparing y's arity of $n = 1$ with g a b's arity of $k = 0$, we determine that one item must be discarded from T. Finally, to avoid traversing twice as many subterms as in the first-order case, we can optimize prefixes: Given a query term $\zeta\ \overline{t}_n$, we can also match prefixes $\zeta\ \overline{t}_k$, where $k < n$, by allowing T to be nonempty when we reach an accepting node.

Similarly to unification and matching, we present finding generalizations in a perfect discrimination tree as a transition system. States are quadruples $Q = (\overline{t}, \overline{b}, D, \sigma)$, where $\overline{t}$ is a list of terms, $\overline{b}$ is a list of tuples storing backtracking information, D is a discrimination (sub)tree, and σ is a substitution.

Let D be a perfect discrimination tree. $\mathcal{T}erm(D)$ denotes the set of terms stored in D. The function $D|_\zeta$ returns the child of D labeled with ζ, if it exists. Child nodes are themselves perfect discrimination (sub)trees. Given any node D, if the node is accepting, then the value stored on that node is defined as $val(D) = (s, d)$, where s is the accepted term and d is some arbitrary data; otherwise, $val(D)$ is undefined.

Starting from an initial state $([t], [\,], D, \emptyset)$, where t is the query term and D is an entire discrimination tree, the following transitions are possible:

AdvanceF $(\mathsf{f}\,\overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) \rightsquigarrow (\overline{s}_m \cdot \overline{t}, (\mathsf{f}\,\overline{s}_m, D, \sigma) \cdot \overline{b}, D|_\mathsf{f}, \sigma)$
if $D|_\mathsf{f}$ is defined

AdvanceX $(s\,\overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) \rightsquigarrow (\overline{s}_m \cdot \overline{t}, (s\,\overline{s}_m, D, \sigma) \cdot \overline{b}, D|_x, \sigma[x \mapsto s])$
if $D|_x$ is defined, x and s have the same type,
and $\sigma(x)$ is either undefined or equal to s

Backtrack $(\overline{s}_m \cdot \overline{t}, (s, D_0, \sigma_0) \cdot \overline{b}, D, \sigma) \rightsquigarrow (s \cdot \overline{t}, \overline{b}, D_0, \sigma_0)$
if $D_0|_\zeta = D$ and $m = arity(\zeta) - arity(s)$

Success $([\,], \overline{b}, D, \sigma) \rightsquigarrow (val(D), \sigma)$
if $val(D)$ is defined

Above, $\cdot$ denotes prepending an element or a list to a list. Intuitively, AdvanceF and AdvanceX move deeper in the tree, generalizing cases A and B above to λfHOL terms. Backtrack can be used to return to a previous state. Success extracts the term t and data d stored in an accepting node.

The following derivation illustrates how to locate a generalization of g (g a b) in the tree D_2:

$$\begin{aligned}
&([\mathsf{g}\ (\mathsf{g}\ \mathsf{a})\ \mathsf{b})], [\,], D, \emptyset)\\
\rightsquigarrow_{\mathsf{AdvanceF}}\ &([\mathsf{g}\ \mathsf{a}\ \mathsf{b}], [(\mathsf{g}\ (\mathsf{g}\ \mathsf{a}\ \mathsf{b}), D, \emptyset)], D|_{\mathsf{g}}, \emptyset)\\
\rightsquigarrow_{\mathsf{AdvanceX}}\ &([\mathsf{b}], [(\mathsf{g}\ \mathsf{a}\ \mathsf{b}, D|_{\mathsf{g}}, \emptyset), \ldots], D|_{\mathsf{g}.y}, \{y \mapsto \mathsf{g}\ \mathsf{a}\})\\
\rightsquigarrow_{\mathsf{AdvanceX}}\ &([\,], [(\mathsf{b}, D|_{\mathsf{g}.y}, \{y \mapsto \mathsf{g}\ \mathsf{a}\}), \ldots], D|_{\mathsf{g}.y.x}, \{y \mapsto \mathsf{g}\ \mathsf{a},\ x \mapsto \mathsf{b}\})\\
\rightsquigarrow_{\mathsf{Success}}\ &((\mathsf{g}\ (y\ x), d), \{y \mapsto \mathsf{g}\ \mathsf{a},\ x \mapsto \mathsf{b}\})
\end{aligned}$$

Let $\rightsquigarrow_{\mathsf{Advance}} = \rightsquigarrow_{\mathsf{AdvanceF}} \cup \rightsquigarrow_{\mathsf{AdvanceX}}$. It is easy to show that Backtrack undoes an Advance transition:

3

Lemma 3.6. *If $Q \rightsquigarrow_{\mathsf{Advance}} Q'$, then $Q' \rightsquigarrow_{\mathsf{Backtrack}} Q$.*

Proof. For both Advance steps, we show that Backtrack restores the state properly. If AdvanceF was applied, we have

$$\begin{aligned}
(\mathsf{f}\,\overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) &\rightsquigarrow_{\mathsf{AdvanceF}} (\overline{s}_m \cdot \overline{t}, (\mathsf{f}\,\overline{s}_m, D, \sigma) \cdot \overline{b}, D|_{\mathsf{f}}, \sigma)\\
&\rightsquigarrow_{\mathsf{Backtrack}} (\overline{t'}, \overline{b}, D, \sigma)
\end{aligned}$$

We must show that $\overline{t'} = \mathsf{f}\,\overline{s}_m \cdot \overline{t}$. Let $k = \mathit{arity}(\mathsf{f})$ and $l = \mathit{arity}(\mathsf{f}\,\overline{s}_m)$. By definition of k, we have $m = k - l$, as in Backtrack's side condition. Thus, $\overline{t'} = \mathsf{f}\,\overline{s}_m \cdot \overline{t}$. The other case is

$$\begin{aligned}
(s\,\overline{s}_m \cdot \overline{t}, \overline{b}, D, \sigma) &\rightsquigarrow_{\mathsf{AdvanceX}} (\overline{s}_m \cdot \overline{t}, (s\,\overline{s}_m, D, \sigma) \cdot \overline{b}, D|_x, \sigma')\\
&\rightsquigarrow_{\mathsf{Backtrack}} (\overline{t'}, \overline{b}, D, \sigma)
\end{aligned}$$

where $\sigma' = \sigma[x \mapsto s]$. Again, we must show that $\overline{t'} = s\,\overline{s}_m \cdot \overline{t}$. Terms x and s must have the same type for AdvanceX to be applicable; therefore, they have the same arity. Then, we conclude $m = \mathit{arity}(s) - \mathit{arity}(s\,\overline{s}_m) = \mathit{arity}(x) - \mathit{arity}(s\,\overline{s}_m)$, as in Backtrack's side condition. Thus, $\overline{t'} = s\,\overline{s}_m \cdot \overline{t}$. □

Lemma 3.7. *If $Q \rightsquigarrow_{\mathsf{Advance}} Q' \rightsquigarrow_{\mathsf{Backtrack}} Q''$, then $Q'' = Q$.*

Proof. By Lemma 3.6, $Q' \rightsquigarrow_{\mathsf{Backtrack}} Q$. Furthermore, Backtrack is clearly functional. Thus, $Q'' = Q$. □

Lemma 3.8. *Let $Q = ([t], [\,], D, \emptyset)$. If $Q \rightsquigarrow^* Q'$, then $Q \rightsquigarrow^*_{\mathsf{Advance}} Q'$.*

Proof. Let $Q = Q_0 \rightsquigarrow \cdots \rightsquigarrow Q_n = Q'$. Let i be the index of the first transition of the form $Q_i \rightsquigarrow_{\mathsf{Backtrack}} Q_{i+1}$. Since Q_0's backtracking stack is empty, we must have $i \neq 0$. Hence, we have $Q_{i-1} \rightsquigarrow_{\mathsf{Advance}} Q_i \rightsquigarrow_{\mathsf{Backtrack}} Q_{i+1}$. By Lemma 3.7, $Q_{i-1} = Q_{i+1}$. Thus, we can shorten the derivation to $Q_0 \rightsquigarrow \cdots \rightsquigarrow Q_{i-1} = Q_{i+1} \rightsquigarrow \cdots \rightsquigarrow Q_n$, thereby eliminating one Backtrack transition. By repeating this process, we eliminate all Backtrack transitions. □

Lemma 3.9. *There exist no infinite chains of the form $Q_0 \rightsquigarrow_{\mathsf{Advance}} Q_1 \rightsquigarrow_{\mathsf{Advance}} \cdots$.*

Proof. With each Advance transition, the height of the discrimination tree decreases by at least one. □

Perfect discrimination trees match a single term against a set of terms. To prove them correct, we will connect them to the transition system $\Longrightarrow$ for matching (Sect. 3.4). This connection will help us show that whenever a discrimination tree stores a generalization

of a query term, this generalization can be found. To express the refinement, we introduce an intermediate transition system, $\hookrightarrow$, that focuses on a single pair of terms (like $\Longrightarrow$) but that solves the constraints in a depth-first, left-to-right fashion and builds the substitution incrementally (like $\rightsquigarrow$). Its initial states are of the form $([s \lesssim^? t], \emptyset)$. Its transitions are as follows:

Decompose	$(\mathsf{f}\ \overline{s}_m \lesssim^? \mathsf{f}\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow ((s_1 \lesssim^? t_1, \ldots, s_m \lesssim^? t_m) \cdot \overline{c}, \sigma)$
DecomposeX	$(x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow ((s_1 \lesssim^? t_1, \ldots, s_m \lesssim^? t_m) \cdot \overline{c}, \sigma[x \mapsto u])$ if x and u have the same type and either $\sigma(x)$ is undefined or $\sigma(x) = u$
Success	$([\,], \sigma) \hookrightarrow \sigma$
Clash	$(\mathsf{f}\ \overline{s}_m \lesssim^? \mathsf{g}\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot$
ClashTypeX	$(x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow \bot$ if x and u have different types
ClashLenXF	$(x\ \overline{s}_m \lesssim^? \mathsf{f}\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot$ if $m > n$
ClashLenXY	$(x\ \overline{s}_m \lesssim^? y\ \overline{t}_n \cdot \overline{c}, \sigma) \hookrightarrow \bot$ if $x \neq y$ and $m > n$
ClashFX	$(\mathsf{f}\ \overline{s} \lesssim^? x\ \overline{t} \cdot \overline{c}, \sigma) \hookrightarrow \bot$
Double	$(x\ \overline{s}_m \lesssim^? u\ \overline{t}_m \cdot \overline{c}, \sigma) \hookrightarrow \bot$ if x and u have the same type, $\sigma(x)$ is defined, and $\sigma(x) \neq u$

We need an auxiliary function to convert $\hookrightarrow$ states to $\Longrightarrow$ states. Let $\alpha(\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}) = \{x_1 \lesssim^? s_1, \ldots, x_m \lesssim^? s_m\}$, $\alpha(\overline{c}, \sigma) = \{c \mid c \in \overline{c}\} \cup \alpha(\sigma)$, and $\alpha(\bot) = \bot$. Moreover, let $\mathcal{S}$ range over states of the form $(\overline{c}, \sigma)$ and $\mathcal{R}$ additionally range over special states of the form σ or $\bot$.

Lemma 3.10. *If $\mathcal{S} \hookrightarrow \mathcal{R}$, then $\alpha(\mathcal{S}) \Longrightarrow^* \alpha(\mathcal{R})$.*

Proof. By case distinction on $\mathcal{R}$. Let $\mathcal{S} = (\overline{c}, \sigma)$.

Case $\mathcal{R} = (\overline{c'}, \sigma')$: Only $\hookrightarrow_{\mathsf{Decompose}}$ and $\hookrightarrow_{\mathsf{DecomposeX}}$ are possible. If $\hookrightarrow_{\mathsf{Decompose}}$ is applied, then $\Longrightarrow_{\mathsf{Decompose}}$ is applicable and results in $\alpha(\mathcal{R})$. If $\hookrightarrow_{\mathsf{DecomposeX}}$ is applied, we have either $m > 0$, and $\Longrightarrow_{\mathsf{DecomposeX}}$ is applicable, or $m = 0$, and $\alpha(\overline{c'}, \sigma') = \alpha(\mathcal{S})$, which implies that the two states are connected by an idle transition of $\Longrightarrow^*$.

Case $\mathcal{R} = \bot$: All the $\hookrightarrow$ rules resulting in $\bot$ except for Double have the same side conditions as the corresponding $\Longrightarrow$ rules. $\hookrightarrow_{\mathsf{Double}}$ corresponds to $\Longrightarrow_{\mathsf{Double}}$ if $m = 0$. If $m \neq 0$, we need an intermediate $\Longrightarrow_{\mathsf{DecomposeX}}$ step before $\Longrightarrow_{\mathsf{Double}}$ can be applied to derive $\bot$. Since $\hookrightarrow_{\mathsf{Double}}$ is applicable, $\sigma(x) = u' \neq u$. Hence, $x \lesssim^? u'$ must be present in $\alpha(\overline{c}, \sigma)$. $\Longrightarrow_{\mathsf{DecomposeX}}$ will augment this set with $x \lesssim^? u$, enabling $\Longrightarrow_{\mathsf{Double}}$.

Case $\mathcal{R} = \sigma$: The only possible rule is $\hookrightarrow_{\mathsf{Success}}$, with $\overline{c} = [\,]$. Since $\alpha(\mathcal{S}) = \alpha(\sigma)$, this transition corresponds to an idle transition of $\Longrightarrow^*$. □

Lemma 3.11. *If $\mathcal{S} \hookrightarrow^! \mathcal{R}$, then $\mathcal{R}$ is either some substitution σ' or $\bot$. If $\mathcal{S} \hookrightarrow^! \sigma'$, then σ' is the MGG of $\alpha(\mathcal{S})$. If $\mathcal{S} \hookrightarrow^! \bot$, then $\alpha(\mathcal{S})$ has no solutions.*

Proof. First, we show that states $\mathcal{S}' = (\overline{c'}, \sigma')$ cannot be normal forms, by exhibiting transitions from such states. If $\overline{c'} = [\,]$, the $\hookrightarrow_{\mathsf{Success}}$ rule would apply. Otherwise, let $\overline{c'} = c_1 \cdot \overline{c''}$ and consider the matching problem $\{c_1\} \cup \alpha(\sigma')$. If this problem is in solved form, c_1 is a constraint corresponding to a solved variable, and we can apply $\hookrightarrow_{\mathsf{DecomposeX}}$ to move the constraint into the substitution. Otherwise, some $\Longrightarrow$ rule can be applied. It necessarily focuses on c_1, since the constraints from $\alpha(\sigma')$ correspond to solved variables. In all cases except for $\Longrightarrow_{\mathsf{DecomposeX}}$, a homologous $\hookrightarrow$ rule can be applied to $\mathcal{S}'$. If $\Longrightarrow_{\mathsf{DecomposeX}}$ would make $\Longrightarrow_{\mathsf{Double}}$ applicable, then we can apply $\hookrightarrow_{\mathsf{Double}}$ to $\mathcal{S}'$; otherwise, $\hookrightarrow_{\mathsf{DecomposeX}}$ is applicable.

Second, by Lemma 3.10, if $\mathcal{S} \hookrightarrow^{!} \sigma'$, then $\alpha(\mathcal{S}) \Longrightarrow^{*} \alpha(\sigma')$. By construction, $\alpha(\sigma')$ is in solved form. Therefore, $\alpha(\mathcal{S}) \Longrightarrow^{!} \alpha(\sigma')$. By completeness of $\Longrightarrow$, the substitution corresponding to $\alpha(\sigma')$—that is, σ'—is the MGG of $\alpha(\mathcal{S})$.

Third, by Lemma 3.10, if $\mathcal{S} \hookrightarrow^{!} \bot$, then $\alpha(\mathcal{S}) \Longrightarrow^{!} \bot$. By soundness of $\Longrightarrow$, $\alpha(\mathcal{S})$ has no solutions. □

Lemma 3.12. *The relation $\hookrightarrow$ is well founded.*

Proof. By Lemma 3.10, every $\hookrightarrow$ transition corresponds to zero or more $\Longrightarrow$ transitions. Since $\Longrightarrow$ is well founded, the only transitions that can violate well-foundedness of $\hookrightarrow$ are the ones that take idle $\Longrightarrow^{*}$ transitions: $\hookrightarrow_{\mathsf{DecomposeX}}$ for $m = 0$ and $\hookrightarrow_{\mathsf{Success}}$. The latter is terminal, so it cannot contribute to infinite chains. As for $\hookrightarrow_{\mathsf{DecomposeX}}$, with $m = 0$, it decreases the following measure μ, which the other rules nonstrictly decrease, with respect to the multiset extension of $<$ on natural numbers: $\mu([s_1 \lesssim^{?} t_1, \ldots, s_m \lesssim^{?} t_m], \sigma) = \{|s_1|, \ldots |s_m|\}$, where $|s|$ denotes the syntactic size of s. □

Lemma 3.13. *If term s generalizes t, then $([s \lesssim^{?} t], \emptyset) \hookrightarrow^{!} \sigma$, where σ is the MGG of $s \lesssim^{?} t$.*

Proof. By Lemma 3.12, there exists a normal form $\mathcal{R}$ starting from $\mathcal{S} = ([s \lesssim^{?} t], \emptyset)$. Since $s \lesssim^{?} t$ is solvable, by Lemma 3.11, and soundness of $\hookrightarrow$ (a consequence of Lemma 3.10 and soundness of $\Longrightarrow$), $\mathcal{R}$ must be the MGG for s and t. □

Lemma 3.14. *If there exists a term $s \in \mathcal{T}erm(D)$ that generalizes the query term t, then there exists a derivation $([t], [\,], D, \emptyset) \rightsquigarrow^{!} ((s, d), \sigma)$.*

Proof. By Lemma 3.13, we know that $(s \lesssim^{?} t, \emptyset) \hookrightarrow^{!} \sigma$ for each $s \in \mathcal{T}erm(D)$ generalizing t. This means that there exists a derivation $([s \lesssim^{?} t], \emptyset) = (\overline{c_0}, \sigma_0) \hookrightarrow \cdots \hookrightarrow (\overline{c_n}, \sigma_n) \hookrightarrow \sigma$. The n first transitions must be Decompose or DecomposeX, and the last transition must be Success.

We show that there exists a derivation of the form $([t], [\,], D, \emptyset) = \mathcal{Q}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n \rightsquigarrow ((s, d), \sigma)$, where $\mathcal{Q}_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. We define $\overline{t_i}$, $\overline{b_i}$, and D_i as follows, for $i > 0$. The list $\overline{t_i}$ consists of the right-hand sides of the constraints $\overline{c_i}$, in the same order. Let *hd* be the function that extracts the head of a list. We set $b_i = (hd(\overline{t_{i-1}}), D_{i-1}, \sigma_{i-1})$. We know that $\overline{c_{i-1}}$ is nonempty, since there exists a transition $(\overline{c_{i-1}}, \sigma_{i-1}) \hookrightarrow (\overline{c_i}, \sigma_i)$; thus, $\overline{t_{i-1}}$ is nonempty. If an accepting node storing s was reached in n steps, the serialization of s must be of the form $\zeta_1 . \cdots . \zeta_n$. Take $D_i = D_{i-1}|_{\zeta_i}$.

The sequence of states $\mathcal{Q}_i$ forms a derivation: If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{Decompose}} (\overline{c}_{i+1}, \sigma_{i+1})$, then $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceF}} \mathcal{Q}_{i+1}$. If $(\overline{c_i}, \sigma_i) \hookrightarrow_{\mathsf{DecomposeX}} (\overline{c}_{i+1}, \sigma_{i+1})$, then $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceX}} \mathcal{Q}_{i+1}$. If $(\overline{c_n}, \sigma_n) \hookrightarrow_{\mathsf{Success}} \sigma$, then $\mathcal{Q}_n \rightsquigarrow_{\mathsf{Success}} ((s, d), \sigma)$. □

Lemma 3.15. *If* $([t],[\,],D,\emptyset) \rightsquigarrow^{+} ((s,d),\sigma)$*, then* $s \in \mathcal{T}erm(D)$ *and* σ *is the MGG of* $s \lesssim^{?} t$*.*

Proof. Let $([t],[\,],D,\emptyset) = \mathcal{Q}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{Q}_n \rightsquigarrow ((s,d),\sigma)$ be a derivation, where $\mathcal{Q}_i = (\overline{t_i}, \overline{b_i}, D_i, \sigma_i)$ for each i. Without loss of generality, by Lemma 3.8, we can assume that the derivation contains no Backtrack transitions.

The first conjunct, $s \in \mathcal{T}erm(D)$, clearly holds for any term found from an initial state. To prove the second conjunct, we first introduce a function *preord* that defines the preorder decomposition of a list of terms: $preord([\,]) = [\,]$ and $preord(\zeta\ \overline{s}_n \cdot \overline{xs}) = (\zeta, \overline{s}_n \cdot \overline{xs}) \cdot preord(\overline{s}_n \cdot \overline{xs})$. Given a term s, $preord([s])$ gives a sequence $(\zeta_1, \overline{u}_1), \ldots, (\zeta_n, \overline{u}_n)$. Since $s \in \mathcal{T}erm(D)$, the sequence $D_0, \ldots, D_n$ follows the preorder serialization of s: $D_i = D_{i-1}|_{\zeta_i}$ for $i > 0$.

Next, we show that there exists a derivation of the form $([s \lesssim^{?} t],\emptyset) = \mathcal{S}_0 \hookrightarrow \cdots \hookrightarrow \mathcal{S}_n \hookrightarrow \sigma$, where $\mathcal{S}_i = (\overline{c_i}, \sigma_i)$. We define $\overline{c_i}$, for $i > 0$, as the list of constraints whose left-hand sides are the elements of $\overline{u_i}$ and right-hand sides are the elements of $\overline{t_i}$, in the order they appear in the respective lists. By inspecting the definition of *preord* and the changes each Advance step makes to the head of $\overline{t_i}$, we can see that $\overline{u_i}$ and $\overline{t_i}$ have the same length. The sequence of states $\mathcal{S}_i$ forms a derivation: If $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceF}} \mathcal{Q}_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{Decompose}} \mathcal{S}_{i+1}$. If $\mathcal{Q}_i \rightsquigarrow_{\mathsf{AdvanceX}} \mathcal{Q}_{i+1}$, then $\mathcal{S}_i \hookrightarrow_{\mathsf{DecomposeX}} \mathcal{S}_{i+1}$. If $\mathcal{Q}_n \rightsquigarrow_{\mathsf{Success}} \sigma$, then $\mathcal{S}_n \hookrightarrow_{\mathsf{Success}} \sigma$. □

Theorem 3.16 (Total Correctness)**.** *Let* D *be a perfect discrimination tree and* t *be a term. The sets* $\{s \in \mathcal{T}erm(D) \mid \exists\sigma.\ \sigma(s) = t\}$ *and* $\{s \mid \exists d, \sigma.\ ([t],[\,],D,\emptyset) \rightsquigarrow^{!} ((s,d),\sigma)\}$ *are equal.*

Proof. This follows from Lemmas 3.14 and 3.15. □

The theorem tells us that given a term t, all generalizations s stored in the perfect discrimination tree can be found, but it does not exclude nondeterminism. Often, both AdvanceF and AdvanceX are applicable. To find all generalizations, we need to follow both transitions. But for some applications, it is enough to find a single generalization.

To cater for both types of applications, E provides iterators that store the state of a traversal. After an iterator is initialized with the root node D and the query term t, each call to FindNextVal will move the iterator to the next node that generalizes the query term and stores a value, indicating an accepting node. After all such nodes have been traversed, the iterator is set to point to *Null*.

The following definitions constitute the high-level interface for iterating through values incrementally or for obtaining all values of nodes that store generalizations of the query term in D.

```
function InitIter(PDTNode D, Term t) is
   i ← Iterator()
   (i.node, i.t_stack, i.t_proc, i.c_iter) ← (D, [t], [ ], Start)
   return i

procedure FindNextVal(Iterator i) is
   do
      FindNextNode(i)
   while i.node ≠ Null ∧
         (¬ i.t_stack.isEmpty() ∨ ¬ i.node.has_val())
```

```
function AllVals(PDTNode D, Term t) is
  i ← InitIter(D, t)
  FindNextVal(i)
  res ← ∅
  while i.node ≠ Null do
    res ← res ∪ {i.node.val()}
    FindNextVal(i)
  return res
```


The core functionality is implemented in FindNextNode, presented below. This procedure moves the iterator to the next node that has not been explored in the search for generalization, or *Null* if the entire tree has been traversed. It first goes through all child nodes labeled with a variable before possibly visiting the child node labeled with a function symbol. The children of a node can be iterated through using a function NextVarChild that, given a tree node and iterator through children, advances the iterator to the child representing the next variable. Furthermore, we assume that the iterator can also be in the distinguished states *Start* and *End*. *Start* indicates that no child has been visited yet; *End* indicates that we have visited all children. Finally, the expression $n.child(\zeta)$ returns a child of the node n labeled ζ if such a child exists or *Null* otherwise.

```
procedure FindNextNode(Iterator i) is
  if i.t_stack.isEmpty() then
    BacktrackToVar(i)
  advanced ← False
  while i.node ≠ Null ∧ ¬ advanced do
    while i.c_iter ≠ End ∧ ¬ advanced do
      i.c_iter ← NextVarChild(i.node, i.c_iter)
      if i.c_iter ≠ End then
        x ← i.c_iter.var()
        t ← i.t_stack.top()
        s ← GobblePrefix(x, t)
        if s ≠ Null ∧
            (x.binding = Null ∨ x.binding = s) then
          i.t_stack.pop()
          for j ← t.num_args
              downto s.num_args + 1 do
            i.t_stack.push(t.args[j])
          if x.binding = Null then
            x.binding ← s
            i.t_proc.push((t, i.node, i.c_iter, True))
          else
            i.t_proc.push((t, i.node, i.c_iter, False))
          i.node ← i.node.child(x)
          advanced ← True
    t ← i.t_stack.top()
```

```
        if i.c_iter = End ∧ ¬ t.head.isVar()
            ∧ D.child(t.head) ≠ Null then
            i.t_stack.pop()
            for j ← t.num_args downto 1 do
                i.t_stack.push(t.args[j])
            i.t_proc.push((t, i.node, End, False))
            i.node ← i.node.child(t.head)
            advanced ← True
        if ¬ advanced then
            BacktrackToVar(i)
        else
            i.c_iter ← Start

    procedure BacktrackToVar(Iterator i) is
      forever do
        if i.t_proc.isEmpty() then
            i.node ← Null
            return
        else
            (t, D, c_iter, var_unbound) ← i.t_proc.pop()
            label_arity ← i.node.label.type.arity
            t_arity ← t.type.arity
            for i ← 1 to label_arity − t_arity do
                i.t_stack.pop()
            i.t_stack.push(t)
            i.node ← D
            i.c_iter ← c_iter
            if var_unbound then
                i.node.label.binding ← Null
            if c_iter ≠ End then
                return
```

The pseudocode uses a slightly different representation of backtracking tuples than $\rightsquigarrow$. In the AdvanceX rule, σ changes only if the variable x was previously not bound. Instead of creating and storing substitutions explicitly, the algorithm simply remembers whether the variable was bound in this step or not, in the *var_unbound* tuple component. Then it relies on the label x of the current node and its *binding* field to carry the substitutions. Similarly, since the strategy is to traverse the tree by first visiting the variable-labeled child nodes, it needs to remember how far it has come with this traversal. This information is stored in the *c_iter* tuple component.

3.5.2 Fingerprint Indices

Fingerprint indices [144] trade perfect indexing for a compact memory representation and more flexible retrieval conditions. The basic idea is to compare terms by looking only at a few predefined sample positions. If we know that term s has symbol f at the head of the

subterm at 2.1 and term t has g at the same position, we can immediately conclude that s and t are not unifiable.

Let A ("at a variable"), B ("below a variable"), and N ("nonexistent") be distinguished symbols not present in the signature, and let $q < p$ denote that position q is a proper prefix of p (e.g., $\varepsilon < 2 < 2.1$). Given a term t and a position p, the *fingerprint function* $\mathcal{G}fpf$ is defined as

$$\mathcal{G}fpf(t,p) = \begin{cases} \mathsf{f} & \text{if } t|_p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|_p \text{ is a variable} \\ \mathsf{B} & \text{if } t|_q \text{ is a variable for some } q < p \\ \mathsf{N} & \text{otherwise} \end{cases}$$

Based on a fixed tuple of positions $\overline{p}_n$, the *fingerprint* of a term t is defined as $\mathcal{F}p(t) = \big(\mathcal{G}fpf(t,p_1),\ldots,\mathcal{G}fpf(t,p_n)\big)$. To compare two terms s and t, it suffices to check that their fingerprints are componentwise compatible using the following unification and matching matrices, respectively:

	f_1	f_2	A	B	N
f_1		✗			✗
f_2	✗				✗
A					✗
B					
N	✗	✗	✗		

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
f_2	✗		✗	✗	✗
A				✗	✗
B					
N	✗	✗	✗	✗	

The rows and columns correspond to s and t, respectively. The metavariables f_1 and f_2 represent arbitrary distinct symbols. Incompatibility is indicated by ✗.

As an example, let $(\varepsilon, 1, 2, 1.1, 1.2, 2.1, 2.2)$ be the sample positions, and let $s = \mathsf{f}(\mathsf{a}, x)$ and $t = \mathsf{f}(\mathsf{g}(x), \mathsf{g}(\mathsf{a}))$ be the terms to unify. Their fingerprints are $\mathcal{F}p(s) = (\mathsf{f}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{B})$ and $\mathcal{F}p(t) = (\mathsf{f}, \mathsf{g}, \mathsf{g}, \mathsf{A}, \mathsf{N}, \mathsf{a}, \mathsf{N})$. Using the left matrix, we compute the compatibility vector $(-, ✗, -, ✗, -, -, -)$. The mismatches at positions 1 and 1.1 indicate that s and t are not unifiable.

A fingerprint index is a trie that stores a term set T keyed by fingerprint. The term $\mathsf{f}(\mathsf{g}(x), \mathsf{g}(\mathsf{a}))$ above would be stored in the node addressed by f.g.g.A.N.a.N, together with other terms that share the same fingerprint. This scheme makes it possible to unify or match a query term s against all the terms T in one traversal. Once a node storing the terms $U \subseteq T$ has been reached, due to overapproximation we must apply unification or matching on s and each $u \in U$.

When adapting this data structure to λfHOL, we must first choose a suitable notion of position in a term. Conventionally, higher-order positions are strings over $\{1, 2\}$, but this is not graceful. Instead, it is preferable to generalize the first-order notion to flattened λfHOL terms—e.g., $x\,\mathsf{a}\,\mathsf{b}|_1 = \mathsf{a}$ and $x\,\mathsf{a}\,\mathsf{b}|_2 = \mathsf{b}$. However, this approach fails on applied variables. For example, although $x\,\mathsf{b}$ and $\mathsf{f}\,\mathsf{a}\,\mathsf{b}$ are unifiable (using $\{x \mapsto \mathsf{f}\,\mathsf{a}\}$), sampling position 1 would yield a clash between b and a. To ensure that positions remain stable under substitution, we propose to number arguments in reverse: $t|^{\varepsilon} = t$ and $\zeta\ t_n \ldots t_1|^{i.p} = t_i|^p$ if $1 \le i \le n$. We use a nonstandard notation, $t|^p$, for this nonstandard notion. The operation is undefined for out-of-bound indices.

Lemma 3.17. *Let s and t be unifiable terms, and let p be a position such that the subterms $s|^p$ and $t|^p$ are defined. Then $s|^p$ and $t|^p$ are unifiable.*

Proof. By structural induction on p. The case $p = \varepsilon$ is trivial.

Case $p = q.i$: Let $s|^q = \zeta\ s_m \ldots s_1$ and $t|^q = \eta\ t_n \ldots t_1$. Since p is defined in both s and t, we have $s|^p = s_i$ and $t|^p = t_i$. By the induction hypothesis, $s|^q$ and $t|^q$ are unifiable, meaning that there exists a substitution σ such that $\sigma(\zeta\ s_m \ldots s_1) = \sigma(\eta\ t_n \ldots t_1)$. Hence, $\sigma(s_1) = \sigma(t_1)$, ..., $\sigma(s_i) = \sigma(t_i)$—i.e., $\sigma(s|^p) = \sigma(t|^p)$. □

Let $t\langle^p$ denote the subterm $t|^q$ such that q is the longest prefix of p for which $t|^q$ is defined. The λfHOL version of the fingerprint function is defined as follows:

$$\mathcal{G}fpf'(t,p) = \begin{cases} \mathsf{f} & \text{if } t|^p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|^p \text{ has a variable head} \\ \mathsf{B} & \text{if } t|^p \text{ is undefined} \\ & \text{but } t\langle^p \text{ has a variable head} \\ \mathsf{N} & \text{otherwise} \end{cases}$$

Except for the reversed numbering scheme, $\mathcal{G}fpf'$ coincides with $\mathcal{G}fpf$ on first-order terms. The fingerprint $\mathcal{F}p'(t)$ of a term t is defined analogously as before, and the same compatibility matrices can be used.

The key difference between $\mathcal{G}fpf$ and $\mathcal{G}fpf'$ concerns applied variables. Given the sample positions $(\varepsilon, 2, 1)$, the fingerprint of x is $(\mathsf{A}, \mathsf{B}, \mathsf{B})$ as before, whereas the fingerprint of $x\ \mathsf{c}$ is $(\mathsf{A}, \mathsf{B}, \mathsf{c})$. As another example, let $(\varepsilon, 2, 1, 2.2, 2.1, 1.2, 1.1)$ be the sample positions, and let $s = x\ (\mathsf{f}\ \mathsf{b}\ \mathsf{c})$ and $t = \mathsf{g}\ \mathsf{a}\ (y\ \mathsf{d})$. Their fingerprints are $\mathcal{F}p'(s) = (\mathsf{A}, \mathsf{B}, \mathsf{f}, \mathsf{B}, \mathsf{B}, \mathsf{b}, \mathsf{c})$ and $\mathcal{F}p'(t) = (\mathsf{g}, \mathsf{a}, \mathsf{A}, \mathsf{N}, \mathsf{N}, \mathsf{B}, \mathsf{d})$. The terms are not unifiable due to the incompatibility at position 1.1 (c vs. d).

We can easily support prefix optimization for both terms s and t being compared: We simply add enough fresh variables as arguments to ensure that s and t are fully applied before computing their fingerprints.

Lemma 3.18. *If terms s and t are unifiable, then $\mathcal{G}fpf'(s,p)$ and $\mathcal{G}fpf'(t,p)$ are compatible according to the unification matrix. If s generalizes t, then $\mathcal{G}fpf'(s,p)$ and $\mathcal{G}fpf'(t,p)$ are compatible according to the matching matrix.*

Proof. We focus on the case of unification. By contraposition, it suffices to consider the eight blank cells in the unification matrix, where the rows correspond to $\mathcal{G}fpf'(s,p)$ and the columns correspond to $\mathcal{G}fpf'(t,p)$. Since unifiability is a symmetric relation, we can rule out four cases.

Case f_1–f_2: By definition of $\mathcal{G}fpf'$, $s|^p$ and $t|^p$ must be of the forms $\mathsf{f}_1\ \overline{s}$ and $\mathsf{f}_2\ \overline{t}$, respectively. Clearly, $s|^p$ and $t|^p$ are not unifiable. By Lemma 3.17, s and t are not unifiable.

Case f_1–N, f_2–N, or A–N: From $\mathcal{G}fpf'(t,p) = \mathsf{N}$, we deduce that $p \neq \varepsilon$. Let $p = q.i.r$, where q is the longest prefix such that $\mathcal{G}fpf'(t,q) \neq \mathsf{N}$. Since $\mathcal{G}fpf'(t,q.i) = \mathsf{N}$, the head of $t|^q$ must be some symbol g. (For a variable head, we would have $\mathcal{G}fpf'(t,q.i) = \mathsf{B}$.) Hence, $t|^q$ has the form $\mathsf{g}\ t_n \ldots t_1$, for $n < i$. Since $q.i$ is a legal position in s, $s|^q$ has the form $\zeta\ s_m \ldots s_1$,

with $i \leq m$. A necessary condition for $\sigma(s|^q) = \sigma(t|^q)$ is that $\sigma(\zeta\ s_m \ldots s_{n+1}) = \sigma(\mathsf{g})$, but this is impossible because the left-hand side is an application (since $n < m$), whereas the right-hand side is the symbol g. By Lemma 3.17, s and t are not unifiable. □

Corollary 3.19 (Overapproximation)**.** *If s and t are unifiable terms, then $\mathcal{F}p'(s)$ and $\mathcal{F}p'(t)$ are compatible according to the unification matrix. If s generalizes t, then $\mathcal{F}p'(s)$ and $\mathcal{F}p'(t)$ are compatible according to the matching matrix.*

3.6 Inference Rules

Saturating provers show the unsatisfiability of a clause set by systematically adding logical consequences, eventually deriving the empty clause as a witness of unsatisfiability. They implement two kinds of inference rules: *Generating rules* produce new clauses and are needed for completeness, whereas *simplification rules* delete existing clauses or replace them by simpler clauses. This simplification is crucial for success, and most modern provers spend a large part of their time on simplification. E's main loop, which applies the rules, implements the given clause procedure, as described in Sect. 2.5.5.

Ehoh is based on the same logical calculus as E, except that it is generalized to λfHOL terms. The standard inference rules and completeness proof of superposition with respect to intensional Boolean-free λfHOL fragment of our logic can be reused verbatim; the only changes concern the basic definitions of terms and substitutions [15, Sect. 1]. Refutational completeness of superposition for λfHOL terms has been formally proved by Peltier [132] using Isabelle. We introduced support for first-class Boolean terms in Ehoh by extending the preprocessor, as explained in Sect. 3.8.

The Generating Rules The rules of the superposition calculus were introduced in Sect. 2.5.3. For completeness, we repeat them with slightly simplified notation, as we do not repeat side conditions:

$$\frac{s \approx t \vee C \qquad u[s'] \not\approx v \vee D}{\sigma(u[t] \not\approx v \vee C \vee D)}\,\mathrm{SN} \qquad \frac{s \not\approx s' \vee C}{\sigma(C)}\,\mathrm{ER}$$

$$\frac{s \approx t \vee C \qquad u[s'] \approx v \vee D}{\sigma(u[t] \approx v \vee C \vee D)}\,\mathrm{SP} \qquad \frac{s \approx t \vee s' \approx u \vee C}{\sigma(t \not\approx u \vee s \approx u \vee C)}\,\mathrm{EF}$$

In each rule, σ denotes the MGU of s and s'.

Equality resolution (ER) and equality factoring (EF) are single-premise rules that work on the entire left- or right-hand side of a literal of the given clause. To generalize them, it suffices to disable prefix optimization for unification.

The rules for superposition into negative and positive literals (SN and SP) are more complex. As two-premise rules, they require the prover to find a partner for the given clause. There are two cases to consider, depending on whether the given clause acts as the first or second premise in an inference. Moreover, since the rules operate on subterms s' of a clause, the prover must be able to efficiently locate all relevant subterms, including λfHOL prefix subterms. To cover the case where the given clause acts as the left premise, the prover relies on a fingerprint index to compute a set of clauses containing

terms possibly unifiable with a side s of a positive literal of the given clause. Thanks to our generalization of fingerprints, in Ehoh this candidate set is guaranteed to overapproximate the set of all possible inference partners. The unification algorithm is then applied to filter out unsuitable candidates. Thanks to prefix optimization, we can avoid polluting the index with all prefix subterms.

When the given clause is the right premise, the prover traverses its subterms s' looking for inference partners in another fingerprint index, which contains only entire left- and right-hand sides of equalities. Like E, Ehoh traverses subterms in a first-order fashion. If prefix unification succeeds, Ehoh determines the unified prefix and applies the appropriate inference instance.

The Simplifying Rules Unlike generating rules, simplifying rules do not necessarily add conclusions to the proof state—they can also remove premises. E implements over a dozen simplifying rules, with unconditional rewriting and clause subsumption as the most significant examples. Here, we restrict our attention to a single rule, which best illustrates the challenges of supporting λfHOL:

$$\frac{s \approx t \qquad u[\sigma(s)] \approx u[\sigma(t)] \vee C}{s \approx t}\,\text{ES}$$

Given an equation $s \approx t$, equality subsumption (ES) removes a clause containing a literal whose two sides are equal except that an instance of s appears on one side where the corresponding instance of t appears on the other side.

E maintains a perfect discrimination tree storing clauses of the form $s \approx t$ indexed by s and t. When applying ES, E considers each positive literal $u \approx v$ of the given clause in turn. It starts by taking the left-hand side u as a query term. If an equation $s \approx t$ (or $t \approx s$) is found in the tree, with $\sigma(s) = u$, the prover checks whether $\sigma'(t) = v$ for some (possibly nonstrict) extension σ' of σ. If so, ES is applicable, with a second premise of the form $\sigma(s) \approx \sigma(t) \vee C$.

To consider nonempty contexts, the prover traverses the subterms u' and v' of u and v in lockstep, as long as they appear under identical contexts. Thanks to prefix optimization, when Ehoh is given a subterm u', it can find an equation $s \approx t$ in the tree such that $\sigma(s)$ is equal to some prefix of u', with some arguments $\overline{u}$ remaining as unmatched. Checking for equality subsumption then amounts to checking that $v' = \sigma'(t)\ \overline{u}$, for some extension σ' of σ.

For example, let $\mathsf{f}\ (\mathsf{g}\ \mathsf{a}\ \mathsf{b}) \approx \mathsf{f}\ (\mathsf{h}\ \mathsf{g}\ \mathsf{b})$ be the given clause, and suppose that $x\ \mathsf{a} \approx \mathsf{h}\ x$ is indexed. Under context $\mathsf{f}\ [\,]$, Ehoh considers the subterms $\mathsf{g}\ \mathsf{a}\ \mathsf{b}$ and $\mathsf{h}\ \mathsf{g}\ \mathsf{b}$. It finds the prefix $\mathsf{g}\ \mathsf{a}$ of $\mathsf{g}\ \mathsf{a}\ \mathsf{b}$ in the tree, with $\sigma = \{x \mapsto \mathsf{g}\}$. The prefix $\mathsf{h}\ \mathsf{g}$ of $\mathsf{h}\ \mathsf{g}\ \mathsf{b}$ matches the indexed equation's right-hand side $\mathsf{h}\ x$ using the same substitution, and the remaining argument in both subterms, b, is identical. Ehoh concludes that the given clause is redundant.

Pragmatic Extensions Since Ehoh is based on a monomorphic logic, the only way to support extensionality without changing the calculus is to add a set of extensionality axioms for every function type occurring in the problem [15, Sect. 3.1]. The evaluation by

Bentkamp et al. of such an approach was discouraging [15, Sect. 6], so we decided to support extensionality via inference rules in Ehoh. We implemented two well-known incomplete rules my colleagues and me had experimented with in the context of Zipperposition.

The negative and positive extensionality (NE and PE) rules are defined as

$$\frac{s \not\approx t \vee C}{s\,(\mathsf{sk}\,\overline{x}) \not\approx t\,(\mathsf{sk}\,\overline{x}) \vee C}\,\mathrm{NE} \qquad \frac{s\,x \approx t\,x \vee C}{s \approx t \vee C}\,\mathrm{PE}$$

For NE, $\overline{x}$ contains all the variables occurring in s and t, the terms s and t are of function type, sk is a fresh Skolem symbol, and the literal $s \not\approx t$ is eligible for resolution [18, Sect. 5]. For PE, variable x does not occur in any of the s, t, or C, no literals are selected in C, and $s\,x \approx t\,x$ is a maximal literal.

Finally, we introduced an injectivity recognition (IR) rule, which detects injectivity axioms and asserts the existence of the inverse function for injective function symbols:

$$\frac{\mathsf{f}\,\overline{x}_n \not\approx \mathsf{f}\,\overline{y}_n \vee x_i \approx y_i}{\mathsf{sk}\,(\mathsf{f}\,\overline{x}_n)\,\overline{x}_J \;\approx\; x_i}\,\mathrm{IR}$$

where sk is a fresh Skolem symbol, and J is the largest subset of $\{1,\ldots,n\}$ such that $x_j = y_j$ for every $j \in J$. We denote the subsequence of $\overline{x}_n$ indexed by J by $\overline{x}_J$. Moreover, we require that $x_i \neq y_i$, all variables in $\overline{x}_K \cdot \overline{y}_K$ are distinct, where $K = \{1,\ldots,n\} \setminus J$, and neither $\overline{x}_K$ nor $\overline{y}_K$ shares variables with $\overline{x}_J$. For example, given $\mathsf{add}\ a\ b \not\approx \mathsf{add}\ a\ b' \vee b \approx b'$, IR can derive the existence of the inverse sk_1 characterized by $\mathsf{sk}_1\,(\mathsf{add}\ a\ b)\ a \approx b$.

3.7 Heuristics

E's heuristics are largely independent of the logic used, and work unchanged for Ehoh. Yet, in preliminary experiments, we noticed that E proved some λfHOL benchmarks quickly using the applicative encoding (Sect. 3.1), whereas Ehoh timed out. There were enough such problems to prompt us to take a closer look. Based on these observations, we extended the heuristics to exploit λfHOL-specific features.

Term Order Generation The superposition calculus is parameterized by a term order—typically an instance of KBO or LPO (Sect. 2.5.1). E can generate a *symbol weight* function (for KBO) and a *symbol precedence* (for KBO and LPO) based on criteria such as the symbols' frequencies, their arities, and whether they appear in the conjecture.

In preliminary experiments, we discovered that the presence of an explicit application operator @ can be beneficial for some problems. Let $\mathsf{a} : \iota_1$, $\mathsf{b} : \iota_2$, $\mathsf{c} : \iota_3$, $\mathsf{f} : \iota_1 \to \iota_2 \to \iota_3$, $x : \iota_2 \to \iota_3$, $y : \iota_2$, and $z : \iota_3$, and consider the clauses $\mathsf{f}\,\mathsf{a}\,y \not\approx \mathsf{c}$ and $x\,\mathsf{b} \approx z$, where the first one is the negated conjecture. Their applicative encoding is $@_{\iota_2,\iota_3}(@_{\iota_1,\iota_2\to\iota_3}(\mathsf{f},\mathsf{a}),y) \not\approx \mathsf{c}$ and $@_{\iota_2,\iota_3}(x,\mathsf{b}) \approx z$, where $@_{\tau,\upsilon}$ is a type-indexed family of symbols representing the application of a function of type $\tau \to \upsilon$. With the applicative encoding, generation schemes can take the symbols $@_{\tau,\upsilon}$ into account, thereby exploiting the type information carried by such symbols. Since $@_{\iota_2,\iota_3}$ is a conjecture symbol, some weight generation scheme could give it a low weight, which would also impact the second clause. By contrast, the native

λfHOL clauses share no symbols; the connection between them is hidden in the types of variables and symbols that are ignored by the heuristics.

To simulate the behavior observed on applicative problems, we introduced four generation schemes that extend E's existing symbol-frequency-based schemes by partitioning the symbols by type. To each symbol, the new schemes assign a frequency equal to the sum of all symbol frequencies for its class. Each new scheme is inspired by a similarly named type-agnostic scheme in E, without `type` in its name:

- `typefreqcount` assigns as each symbol's weight the number of occurrences of symbols of the same type.
- `typefreqrank` sorts the frequencies calculated by the function `typefreqcount` in increasing order and assigns each symbol a weight corresponding to its rank.
- `invtypefreqcount` is `typefreqcount`'s inverse. If `typefreqcount` would assign a weight w to a symbol, it assigns $M - w + 1$, where M is the maximum symbol weight according to `typefreqcount`.
- `invtypefreqrank` is `typefreqrank`'s inverse. It sorts the frequencies in decreasing order.

We designed four more schemes (whose names begin with `comb` instead of `type`) that combine E's type-agnostic and Ehoh's type-aware approaches using a linear equation.

To generate symbol precedences, E can sort symbols by weight and use the symbol's position in the sorted array as the basis for precedence. To reflect the type information introduced by the applicative encoding, we implemented four type-aware precedence generation schemes. Ties are broken by comparing the symbols' number of occurrences and, if necessary, the position of their first occurrence in the input.

Literal Selection The side conditions of the superposition rules SN and SP (Sect. 2.5.3) rely on a literal selection function to restrict the set of *inference literals*, thereby reducing the search space. Given a clause, a literal selection function returns a (possibly empty) subset of its literals. For completeness, any nonempty subset selected must contain at least one negative literal. If no literal is selected, all *maximal* literals become inference literals. The selection function E uses most often is probably `SelectMaxLComplexAvoidPosPred`, which we abbreviate to `SelectMLCAPP`. It selects at most one negative literal, based on size, absence of variables, and maximality of the literal in the clause.

Intuitively, applied variables can potentially be unified with more terms than terms with rigid heads. This makes them prolific in terms of possible inference partners, a behavior that can lead to creating many unnecessary clauses and should thus be avoided. On the other hand, shorter proofs might be found if we prefer selecting applied variables. To cover both scenarios, we implemented selection functions that prefer or defer selecting applied variables.

Let $\mathit{max}(L) = 1$ if L is a maximal literal of the clause it appears in; otherwise, $\mathit{max}(L) = 0$. Let $\mathit{appvar}(L) = 1$ if L is a literal where either side is an applied variable; otherwise, $\mathit{appvar}(L) = 0$. Based on these definitions, we devised the following selection functions, both of which rely on `SelectMLCAPP` to break ties:

- `SelectMLCAPPAvoidAppVar` selects a negative literal L with the maximal value of $(max(L), 1 - appvar(L))$ according to the lexicographic order.

- `SelectMLCAPPPreferAppVar` selects a negative literal L with the maximal value of $(max(L), appvar(L))$ according to the lexicographic order.

Clause Selection Selection of the given clause is a critical choice point. E heuristically assigns *clause priorities* and *clause weights* to the candidates. The priorities provide a crude partition, whereas the weights order the clauses within a partition. E's main loop visits a set of priority queues in round-robin fashion. From a given queue, the clause with the highest priority and the smallest weight is selected. Typically, one of the queues will use the clauses' age as priority, to ensure fairness (i.e., that each nonredundant clause is eventually chosen).

E provides template weight functions that allow users to fine-tune parameters such as weights assigned to variables or function symbols. The most widely used template is `ConjectureRelativeSymbolWeight`, which we abbreviate to `CRSWeight`. It computes term and clause weights according to eight parameters, notably *conj_mul*, a multiplier applied to the weight of conjecture symbols. This template works well for some applicatively encoded problems. Let $\mathsf{a} : \iota$, $\mathsf{f} : \iota \to \iota$, $x : \iota$, and $y : \iota \to \iota$, and consider the clauses $y\ x \not\approx x$ and $\mathsf{f}\ \mathsf{a} \approx \mathsf{a}$, where the first one is the negated conjecture. Their encoding is $@_{\iota,\iota}(y, x) \not\approx x$ and $@_{\iota,\iota}(\mathsf{f}, \mathsf{a}) \approx \mathsf{a}$. The encoded clauses share $@_{\iota,\iota}$, whose weight will be multiplied by *conj_mul*—usually a factor in the interval $(0, 1)$. By contrast, the native λfHOL clauses share no symbols, and the heuristic would fail to notice that f and y have the same type, giving a higher weight to the second clause. To mitigate this, we coded a new type-aware template, `CRSTypeWeight`, that applies the *conj_mul* multiplier to all symbols whose type occurs in the conjecture. For the example above, since $\iota \to \iota$ appears in the conjecture, it would notice the relation between the conjecture variable y and the symbol f and multiply f's weight by *conj_mul*.

Natively supporting λfHOL allows the prover to recognize applied variables. It may make sense to extend clause weight templates to either penalize or promote clauses with such variables. To support this extension, we added the following parameter to `CRSTypeWeight`, as well as to some other E's weight function templates: *appv_mul* is a multiplier applied to terms $s = x\ \overline{t}_n$, where s is either side of the literal and $n > 0$. In addition, we implemented a new clause priority scheme, `ByAppVarNum`, that separates the clauses by the number of top-level applied variables occurring in the clause, favoring those containing fewer such variables.

Configurations and Modes A combination of parameters, including term order, literal selection, and clause selection, is called a *configuration*. For years, E has provided an *auto* mode that analyzes the input problem and chooses a configuration known to perform well on similar problems. More recently, E has been extended with an *autoschedule* mode that applies a portfolio of configurations in sequence on the given problem, restarting the prover for each configuration.

Configurations that are suitable for a wide range of problems have emerged over time. One of them is the configuration that is most often chosen by E's *auto* mode when running on TPTP benchmarks. We call it *boa* ("best of *auto*"):

Term order: KBO

Weight generation: `invfreqrank`

Precedence generation: `invfreq`

Literal selection: `SelectMLCAPP`

Clause selection:

```
1.CRSWeight(SimulateSOS, 0.5, 100, 100, 100, 100, 1.5, 1.5, 1),
4.CRSWeight(ConstPrio, 0.1, 100, 100, 100, 100, 1.5, 1.5, 1.5),
1.FIFOWeight(PreferProcessed),
1.CRSWeight(PreferNonGoals, 0.5, 100, 100, 100, 100, 1.5, 1.5, 1),
4.Refinedweight(SimulateSOS, 3, 2, 2, 1.5, 2)
```

The clause selection scheme consists of five queues, each of which is specified by a weight function template. The prefixes $n.$ next to the template names indicate that the queue will be visited n times in the round-robin scheme before moving to the next one. The first argument to each template is the clause priority scheme.

3.8 Preprocessing

E's preprocessor transforms first-order formulas into clausal normal form, before the main loop is started. As explained in Sect. 2.4, E also encodes all literals as equations. Beyond turning the problem into a conjunction of disjunctive clauses, the preprocessor eliminates quantifiers, introducing Skolem symbols for essentially existential quantifiers.

For first-order logic, skolemization preserves both satisfiability (unprovability) and unsatisfiability (provability). In contrast, for higher-order logics without the axiom of choice, naive skolemization is unsound, because it introduces symbols that can be used to instantiate higher-order variables. One solution proposed by Miller [119, Sect. 6] is to ensure that Skolem symbols are always applied to a minimal number of arguments. However, to keep the implementation simple, we have decided to ignore this issue and consider all arguments as optional, including those to Skolem symbols. In Chapter 7 we extend Ehoh's logic to full higher-order logic with the axiom of choice, which addresses the issue.

There is another transformation performed by preprocessing that is problematic, but for a different reason. *Definition unfolding* is the process of replacing equationally defined symbols with their definitions and removing the defining equations. A definition is a clause of the form $\mathsf{f}\,\overline{x}_m \approx t$, where the variables $\overline{x}_m$ are distinct, f does not occur in the right-hand side t, and $\mathcal{V}ar(t) \subseteq \{x_1, \ldots, x_m\}$. This transformation preserves unsatisfiability (provability) for first- and higher-order logic, but not for λfHOL, making Ehoh incomplete. The reason is that by removing the definitional clause, we also remove a symbol f that otherwise could be used to instantiate a higher-order quantifier. For example, the clause set

$\{\mathsf{f}\ x \approx x, \mathsf{f}\ (y\ \mathsf{a}) \not\approx \mathsf{a}\}$ is unsatisfiable, whereas $\{y\ \mathsf{a} \not\approx \mathsf{a}\}$ is satisfiable in λfHOL. (In full higher-order logic, the second clause set would be unsatisfiable thanks to the $\{y \mapsto \lambda x.\ x\}$ instance and β-reduction.) For the moment, we have simply disabled definition unfolding in Ehoh. We have not measured the effect of this choice, but we conjecture it is not substantial. Appropriately chosen term order will result in a similar outcome without removing the defined symbol. We will enable definition unfolding again once we have added support for λ-terms (Chapter 7).

Higher-order logic treats formulas as terms of Boolean type, erasing the distinction between terms and formulas. As a consequence, formulas might appear as arguments not only to logical connectives but also to function symbols or applied variables—e.g., $\mathsf{p}\,(\mathsf{a} \wedge \mathsf{b})$, $y\,(\neg\,\mathsf{a})$. We call such formulas *nested*. Kotelnikov et al. [98] describe a modification to Vampire's clausification algorithm to support nested formulas. We adapt their approach to the clausification algorithm [126] used by E. Given a formula φ to clausify, the following procedure removes nested formulas:

1. Let $\chi = \varphi|_p$ be the leftmost outermost nested formula that is different from $\top$, $\bot$, or a variable x, if one exists; otherwise, skip to step 2. Let $p = q.r$ where q is the longest strict prefix of p such that $\psi = \varphi|_q$ is a formula. Let $\psi' = (\chi \longrightarrow \psi[\top]_r) \wedge (\neg\chi \longrightarrow \psi[\bot]_r)$. Replace φ by $\varphi[\psi']_q$ and repeat this step.

2. Apply all the steps of E's clausification algorithm up to and including skolemization.

3. Skolemization might replace Boolean variables by new terms with predicate symbol heads. To remove them, follow step 1.

4. Perform the remaining steps of E's clausification algorithm, resulting in a set of clauses.

5. Let C be a clause that contains a literal L of the form $x \approx \top$ or $x \not\approx \top$, where x is a Boolean variable, if one exists; otherwise, terminate. Delete C if it also contains the complement of L. Otherwise, replace C with the clause $C[x \mapsto \bot]$ if L is of the form $x \approx \top$ and else $C[x \mapsto \top]$. Trivial literals $\bot \approx \top$ and $\top \not\approx \top$ are removed from the resulting clause. Repeat this step.

As an example, consider the formula $\mathsf{f}\ x \not\approx x \longrightarrow \mathsf{p}\ (\mathsf{a} \wedge \mathsf{b})$. Step 1 moves the subterm $\mathsf{a} \wedge \mathsf{b}$ outward, yielding $\mathsf{f}\ x \not\approx x \longrightarrow ((\mathsf{a} \wedge \mathsf{b}) \longrightarrow \mathsf{p}\,\top) \wedge (\neg\,(\mathsf{a} \wedge \mathsf{b}) \longrightarrow \mathsf{p}\,\bot)$. This formula can be clausified further as usual.

Theorem 3.20 (Total Correctness)**.** *The above procedure always terminates and produces a set of clauses that is equisatisfiable with the original formula φ in λfHOL with interpreted Booleans and that contains no nested formulas other than $\top$, $\bot$, and variables.*

Proof. It is easy to see that steps 1, 3, and 5 produce equivalent formulas or clauses. Moreover, steps 1 and 3 remove all offending nested formulas (i.e., other than $\top$, $\bot$, and variables). In conjunction with the standard clausification algorithm, which preserves and reflects satisfiability, our procedure gives correct results when it terminates.

To prove termination, we will use a measure function $\mathcal{W}$ to natural numbers that decreases with each application of step 1 or 3. Steps 2 and 4 rely on a terminating algorithm, whereas each application of step 5 decreases the size of a clause. We define $\mathcal{W}$ by

$\mathcal{W}(\forall x.\, s) = \mathcal{W}(\exists x.\, s) = \mathcal{W}(s)$; $\mathcal{W}(\zeta\, \overline{s}_n) = \sum_{i=1}^{n} \mathcal{W}(s_i)$ if ζ is a logical connective (including $\top$ and $\bot$); and $\mathcal{W}(\zeta\, \overline{s}_n) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$ otherwise, where k is the number of offending outermost nested formulas in $\zeta\, \overline{s}_n$. We must show $\mathcal{W}(\psi) > \mathcal{W}(\psi')$. By definition, ψ is of the form $\zeta\, \overline{s}_n$, where ζ is not a logical connective. Thus $\mathcal{W}(\psi) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$. Steps 1 and 3 substitute $\top$ or $\bot$, of measure 0, for a nested formula χ (including χ's own nested formulas) in ψ. Clearly, the longer r is, the more $\mathcal{W}(\psi')$ decreases. Taking $|r| = 1$, we get the upper bound $2\mathcal{W}(\chi) + 2 \cdot 3^{k-1}(1 + \sum_{i=1}^{n} \mathcal{W}(s_i) - \mathcal{W}(\chi))$ for $\mathcal{W}(\psi')$, which is less than $\mathcal{W}(\psi) = 3^k(1 + \sum_{i=1}^{n} \mathcal{W}(s_i))$. □

The output may contain $\top$, $\bot$, or Boolean variables as nested formulas. Since E was first developed as an untyped prover, unification of a variable with a Boolean constant was disallowed to avoid unsoundness. We needed to undo this in Ehoh. Ehoh also removes trivial literals $\bot \approx \top$ and $\top \not\approx \top$ that emerge during proof search.

3.9 Evaluation

How useful are Ehoh's new heuristics? And how does Ehoh perform compared with E, used directly or in tandem with the applicative encoding, and compared with other provers? To answer the first question, we evaluated each new heuristic scheme independently. From the empirical results, we derived a new configuration optimized for λfHOL. For the second question, we compared Ehoh's success rate and speed on λfHOL problems with native higher-order provers and on applicatively encoded problems with E. We also included first-order benchmarks to measure Ehoh's overhead.

We set a CPU time limit of 60 s per problem. This is more than allotted by interactive proof tools such as Sledgehammer, or by cooperative provers such as Leo-III and Satallax, but less than the 300 s of CASC [159]. The experiments were performed on StarExec [154] nodes equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz.

Heuristics Tuning We used the *boa* configuration as the basis to evaluate the new heuristic schemes. For each heuristic parameter we tuned, we changed only its value while keeping the other parameters the same as for *boa*. This gave an idea of how each parameter affected overall performance. All heuristic parameters were tested on a 5012 problem suite generated using Sledgehammer, consisting of four variants of the Judgment Day [34] suite. The problems were given in native λfHOL syntax. The experiments described in this subsection were carried out using an earlier E version (2.3).

Evaluating the new weight and precedence generation heuristics amounted to testing each possible combination of frequency-based schemes, including E's original type-agnostic schemes. Figure 3.1 shows the number of solved (i.e., proved or disproved) problems for each combination. In this and the following figures, the underlined number is for *boa*, whereas bold singles out the best value. In the names of the generation schemes, we abbreviated `inv` to `i`, `type` to `t`, `freq` to `f`, `comb` to `cm`, `count` to `cn`, and `rank` to `r`.

Figure 3.1 indicates that including type information in the generation schemes results in a somewhat higher number of solved problems compared with E's type-agnostic schemes. Against our expectations, Ehoh's combined schemes appear to be less efficient than the type-aware schemes.

	f	if	tf	itf	cmf	icmf
fcn	2294	2288	2287	2297	2290	2287
ifcn	2371	2373	2374	2370	2369	2377
fr	2326	2317	2323	2329	2322	2318
ifr	2383	2379	2376	2380	2381	2381
tfcn	2305	2314	2301	2306	2302	2311
itfcn	2386	2381	2389	2388	2384	2379
tfr	2326	2334	2322	2334	2321	2336
itfr	2390	2382	2390	**2394**	2387	2386
cmfcn	2273	2281	2271	2285	2269	2280
icmfcn	2380	2375	2382	2379	2380	2375
cmfr	2321	2313	2319	2321	2318	2312
icmfr	2368	2378	2371	2378	2368	2380

Figure 3.1: Evaluation of weight and precedence generation schemes

	0.25	0.35	0.5	0.7	1	1.41	2	2.82	4
W	2311	2341	2363	2374	**2379**	2376	2377	2376	2377
TW	2331	2331	2360	2371	2372	2374	2373	2373	2372

Figure 3.2: Evaluation of weight function and *appv_mult* factor

The literal selection function has little impact on performance: Ehoh solves 2379 problems with `SelectMLCAPP` or `SelectMLCAPPAvoidAppVar`, and one less with `SelectMLCAPPPreferAppVar`.

Clause selection is the heuristic component that we extended the most. We had to assess the effect of a new heuristic weight function, a multiplier for the occurrence of top-level applied variables, and clause priority based on the number of top-level applied variables.

To test the effect of the new type-based weight function, we replaced *boa*'s queue that uses `4.CRSWeight(...)`, with the queue ordered by `4.CRSTypeWeight(...)`. The original heuristic is called `W` and the type-aware alternative `TW`. We intuitively chose nine values for testing the effect of the applied variable multiplier *appv_mult*. Figure 3.2 summarizes the results of combining `W` or `TW` with the different *appv_mult* values. Applying a multiplier smaller than 1, which corresponds to preferring literals containing applied variables, can lose dozens of solutions. Overall, using the type-aware heuristic seems slightly detrimental.

Finally, we evaluated the new clause priority function `ByAppVarNum`, by replacing `4.CRSWeight(ConstPrio,...)` with `4.CRSWeight(ByAppVarNum,...)` in *boa*'s specification. `ConstPrio` assigns each clause the same priority. The results are inconclusive.

The results presented above give an idea of how each parameter influences performance. We also evaluated their performance in combination, to derive an alternative to *boa* for λfHOL. For each category of parameters, we chose either *boa*'s value of the parameter in *boa* ("Old") or the best performing newly implemented parameter ("New"). Based on the results above, for term orders, we chose the combination of `invtypefreqrank` and

Term order	Literal selection	Clause weight	Solved
Old	Old	Old	2379
Old	Old	New	2374
Old	New	Old	2379
Old	New	New	2373
New	Old	Old	2394
New	Old	New	**2397**
New	New	Old	2395
New	New	New	**2397**

Figure 3.3: Evaluation of combinations of new parameters

`invtypefreq`; for clause selection, we chose `CRSTypeWeight` with `ConstPrio` priority and an *appv_mult* factor of 1.41; for literal selection, we chose `SelectMLCAPPAvoidAppVar`.

Figure 3.3 shows the number of solved problems for all combinations of these parameters. From the two configurations that solve 2397 problems, we selected the "New Old New" combination as our suggested "higher-order best of *auto*," or *hoboa*, configuration.

Main Evaluation We now present a more detailed evaluation of *hoboa*, along with other configurations, on a larger benchmark suite. Our raw data are publicly available.[2]

The benchmarks are divided into four sets: (1) 1147 first-order TPTP [157] problems belonging to the FOF (untyped) and TF0 (monomorphic) categories, excluding arithmetic; (2) 5012 Sledgehammer-generated problems from the Judgment Day [34] suite, targeting the monomorphic first-order logic embodied by TPTP TF0; (3) all 955 monomorphic higher-order problems from the TH0 category of the TPTP belonging to our extension of λfHOL; (4) 5012 Judgment Day problems targeting the λfHOL fragment of TPTP TH0.

The TPTP includes benchmarks from various areas of computer science and mathematics. It is the de facto standard for evaluating automatic provers, but it has few higher-order problems. For the first group of benchmarks, we randomly selected 1000 FOF problems (out of 8172) and all monomorphic TFF problems that are parsable by E within 60 s (amounting to 147 out of 231 monomorphic TFF problems). Both groups of Sledgehammer problems include two subgroups of 2506 problems, generated to include 32 or 512 Isabelle lemmas (SH32 and SH512), to represent both small and large problems. Each subgroup consists of two sub-subgroups of 1253 problems, generated by using either λ-lifting or SKBCI-style combinators to encode λ-expressions.

To ascertain the effectiveness of our approach, we evaluated Ehoh against E used on applicative encodings of problems (denoted by @+E). For reference, we also evaluated the following versions of higher-order provers that competed in the THF division of the 2019 edition of CASC [160]: CVC4 1.8 prerelease [11], Leo-III 1.4 [153], Satallax 3.4 [39], Vampire 4.4 [24], and Zipperposition 1.6 [48, 49]. Like at CASC, we used different versions of Vampire for first-order and higher-order problems. Similarly, Zipperposition does not use E as backend when it is run on first-order problems and uses different heuristics on first- and higher-order problems. The genuine higher-order provers have the advantage that they

[2] `https://doi.org/10.5281/zenodo.4045452`

	First-order			Higher-order		
	TPTP	SH32	SH512	TPTP	SH32	SH512
E a	624	938	1237			
E as	**665**	**957**	**1298**			
E b	550	943	1242			
@+E a	531	932	1111	686	952	1125
@+E as	571	949	1148	692	969	1164
@+E b	536	943	1227	690	959	1267
Ehoh a	624	939	1236	694	966	1235
Ehoh as	**665**	**957**	1296	**699**	**988**	**1309**
Ehoh b	550	943	1242	697	967	1262
Ehoh hb	504	947	1231	693	975	1267
CVC4	567	956	1361	745	973	**1351**
Leo-III	548	960	1239	**834**	967	1266
Vampire	**728**	**968**	**1401**	805	979	1214
Satallax				827	871	1019
Zipperposition	496	933	1187	815	976	1069

Figure 3.4: Number of proved problems

can instantiate higher-order variables with λ-terms. Thus, some formulas that are provable by these systems may be nontheorems for @+E and Ehoh, or they may require tedious reasoning about λ-lifted functions or SKBCI-style combinators. An example is the conjecture $\exists f.\, \forall x\, y.\ f\ x\ y \approx \mathsf{g}\ y\ x$, whose proof requires taking $\lambda x\, y.\ \mathsf{g}\ y\ x$ as the witness for f.

We ran all provers except Satallax (which supports only THF) on first-order benchmarks to measure the overhead introduced by our extensions, as well as that entailed by the applicative encoding. Figure 3.4 gives the number of problems each system proved. In each column, bold highlights the best E value and the best value overall. We considered the E modes *auto* (a) and *autoschedule* (as) and the configurations *boa* (b) and *hoboa* (hb).

We observe the following. First, comparing the Ehoh row with the E row, we see that Ehoh's overhead is barely noticeable—the difference is at most two problems. Second, Ehoh outperforms the applicative encoding on both first-order and higher-order problems. Nevertheless, the raw evaluation data reveal that there are quite a few higher-order problems that @+E proves faster than Ehoh. Third, it is advantageous to use the higher-order versions of the Sledgehammer problems, although the difference in success rate is small, especially for SH512. Fourth, the new *hoboa* outperforms *boa* on most higher-order problems, suggesting that it could be worthwhile to re-train *auto* and *autoschedule* based on λfHOL benchmarks and to design further heuristics. Fifth, Ehoh cannot compete against the best higher-order systems, but this is no surprise given that it does not support λ-expressions and higher-order unification. An extension of Ehoh that supports these features is described in Chapter 7.

Next to the success rate, the time in which a prover gives an answer is also an important consideration. Figure 3.5 compares the average running times, in seconds, of the various systems on the problems that all of the applicable systems proved. Clearly, Ehoh incurs

little overhead on first-order problems. The raw evaluation data reveal that for *boa*, it takes Ehoh 2747 s to prove all first-order problems that E, @+E, and Ehoh can all prove using this configuration, compared with 2728 s for E, amounting to a 0.7% overhead. For comparison, @+E needs 3939 s—a 44% overhead.

	First-order			Higher-order		
	TPTP	SH32	SH512	TPTP	SH32	SH512
E a	0.22	0.15	0.54			
E as	0.38	0.20	0.74			
E b	0.43	**0.07**	0.56			
@+E a	0.61	0.18	**0.38**	0.03	0.21	**0.32**
@+E as	0.91	0.18	0.39	0.06	0.25	0.33
@+E b	0.53	0.12	0.81	0.09	0.20	0.54
Ehoh a	**0.21**	0.15	0.54	0.03	0.08	0.51
Ehoh as	0.38	0.20	0.73	0.07	0.14	0.60
Ehoh b	0.42	**0.07**	0.58	**0.02**	**0.07**	0.37
Ehoh hb	0.69	0.12	1.06	0.10	0.13	0.56
CVC4	3.02	1.58	1.75	1.22	2.44	1.65
Leo-III	1.33	0.52	5.63	0.49	0.89	6.54
Vampire	0.67	0.43	1.50	0.76	1.89	4.84
Satallax				2.45	5.22	10.12
Zipperposition	3.81	1.60	5.09	0.76	2.21	6.31

Figure 3.5: Average running times on the problems proved by all systems

3.10 Discussion and Related Work

Our working hypothesis is that it is possible to extend first-order provers to higher-order logic without slowing them down unduly. Our research program is two-pronged: On the theoretical side, we are investigating higher-order extensions of superposition [15, 18, 170]; on the practical side, we are implementing such extensions in a state-of-the-art prover. In this thesis, the focus is on the second aspect.

The work described in this chapter was large in scope: it required modifying almost all parts of the E prover, from the parser, to the inference engine, to heuristics. The invariant that variables cannot be applied and that symbols are always passed the same number of arguments were entrenched in E's code, requiring hundreds of modifications. Nonetheless, we found the generalization manageable. The generalization put us in a position to add support for λ-terms and higher-order unification, as discussed in Chapter 7.

Traditionally, most higher-order provers were designed from the ground up to target higher-order logic. Two exceptions are Otter-λ by Beeson [14] and Zipperposition by Cruanes et al. [48, 49]. Otter-λ adds λ-terms and second-order unification to the first-order prover Otter [115]. Zipperposition, based on superposition, was extended to Boolean-free higher-order logic by Bentkamp et al. [18]. Its performance is a far cry from E's, but it is easier to modify. Vukmirović et al. also used it to test and evaluate higher-order

unification procedures [168] and Boolean reasoning [170]. Zipperposition now includes Ehoh as a backend in a cooperative architecture. Finally, there is work by the developers of Vampire [24] and of the SMT solvers CVC4 and veriT [11] to extend their provers to higher-order logic.

Native higher-order reasoning was pioneered by Robinson [140], Andrews [1], and Huet [81]. Andrews [2] and Benzmüller and Miller [20] provide excellent surveys. TPS, by Andrews et al. [3], was based on expansion proofs and lets users specify proof outlines. The Leo family of systems, developed by Benzmüller and his colleagues, is based on resolution and paramodulation. LEO [19] supported extensionality on the calculus level and introduced the cooperative paradigm to integrate first-order provers. Leo-III [153] expands the cooperation with SMT solvers and introduces term orders in a pragmatic, incomplete way. Brown's Satallax [39] is based on a complete higher-order tableau calculus, guided by a SAT solver; later versions also cooperate with E and Ehoh. Another noteworthy system is Lindblad's agsyHOL [107]. It is based on a focused sequent calculus driven by a generic narrowing engine.

An alternative to all of the above is to reduce higher-order logic to first-order logic via a translation. Robinson [141] outlined this approach decades before tools such as MizAR [166], Sledgehammer [131], HOLyHammer [89], and CoqHammer [51] popularized it in proof assistants. In addition to performing an applicative encoding, such translations must eliminate the λ-expressions [50, 117] and encode the type information [29]. In practice, on problems with a large first-order component, translations perform very well compared with native provers [156]. Largely thanks to Sledgehammer, Isabelle often came in a close second at CASC, even defeating Satallax in 2012 [158].

By removing the need for the applicative encoding, our work reduces the translation gap. The applicative encoding buries the λfHOL terms' heads under layers of @ symbols. Terms double in size, cluttering the data structures, and twice as many subterm positions must be considered for inferences. Moreover, the encoding is incompatible with interpreted operators, notably for arithmetic. A common remedy is to introduce proxies to connect an uninterpreted nullary symbol with its interpreted counterpart (e.g., $@(@(\mathsf{add}, x), y) \approx x + y$), but this is clumsy. A further complication is that in a monomorphic logic, @ is not a single symbol but a family of symbols $@_{\tau,\upsilon}$, which must be correctly introduced and recognized. Finally, the encoding must be undone in the proofs. While it should be possible to base a higher-order prover on such an encoding, the prospect is aesthetically and technically unappealing, and performance would likely suffer.

3.11 Conclusion

Despite considerable progress since the 1970s, until a few years ago higher-order automated reasoning had not assimilated some of the most successful methods for first-order logic with equality, such as superposition. We presented a graceful extension of a state-of-the-art first-order theorem prover to a fragment of higher-order logic devoid of λ-terms. Our work covers both theoretical and practical aspects. Experiments show promising results on λ-free higher-order problems and very little overhead for first-order problems, as we would expect from a graceful generalization.

Despite its lack of support for λ-terms, Ehoh is already deployed as a backend in the leading higher-order provers Satallax and Zipperposition. Ehoh also forms the basis of

our work toward stronger higher-order automation. Our aim is to turn it into a prover that excels on proof obligations arising in interactive verification, which tend to be large but only mildly higher-order [156]. In Chapter 7 we describe the extension of Ehoh to full higher-order logic. It was heavily inspired by the techniques implemented in Zipperposition, which helped it dominate CASC in 2020 and 2021.

4

Efficient Full Higher-Order Unification

Joint work with
Alexander Bentkamp and Visa Nummelin

We developed a procedure to enumerate complete sets of higher-order unifiers based on work by Jensen and Pietrzykowski. Our procedure removes many redundant unifiers by carefully restricting the search space and tightly integrating decision procedures for fragments that admit a finite complete set of unifiers. We identify a new such fragment and describe a procedure for computing its unifiers. Our unification procedure, together with new higher-order term indexing data structures, is implemented in the Zipperposition theorem prover. Experimental evaluation shows a clear advantage over Jensen and Pietrzykowski's procedure.

In this work I designed the main algorithm, solid oracle, and fingerprint indexing. I also implemented and evaluated the algorithms. Visa Nummelin proved the completeness of the main algorithm with Alexander Bentkamp's help. The completeness and termination of solid oracle was proven by me with Alexander Bentkamp's help.

4.1 Introduction

As mentioned in Chapter 1, many of the reasoning systems deployed in various areas of computer science and mathematics are based on variants of higher-order logic. More often than not, they also require unification of higher-order terms. Due to its undecidability and explosiveness, the higher-order unification problem is considered one of the main obstacles on the road to efficient higher-order tools.

One of the reasons for higher-order unification's explosiveness lies in *flex-flex pairs*, which consist of two variable-headed terms, e.g., $F\,X \stackrel{?}{=} G\,\mathsf{a}$. Even this seemingly simple problem has infinitely many incomparable unifiers. One of the first methods designed to combat this explosion is Huet's preunification [82]. Huet noticed that some logical calculi would remain complete if flex-flex pairs are not eagerly solved but postponed as constraints. If only flex-flex constraints remain, we know that a unifier must exist and we do not need to solve them. Huet's preunification has been used in many reasoning tools including Isabelle [125], Leo-III [153], and Satallax [39]. However, recent developments in higher-order theorem proving [18, 24] require full unification—i.e., enumeration of unifiers even for flex-flex pairs, which is the focus of this chapter.

Jensen and Pietrzykowski's (JP) procedure [86] is the best known procedure for this purpose (Sect. 4.2). Given two terms to unify, it first identifies a position where the terms disagree. Then, in parallel branches of the search tree, it applies suitable substitutions, involving a variable either at the position of disagreement or above, and repeats this process on the resulting terms until they are equal or trivially nonunifiable.

Building on the JP procedure, we designed a new procedure (Sect. 4.3) with the same completeness guarantees (Sect. 4.4). The new procedure addresses many of the issues that are detrimental to the performance of the JP procedure. First, the JP procedure does not terminate in many cases of obvious nonunifiability, e.g., for $X \stackrel{?}{=} \mathsf{f}\,X$, where X is a nonfunctional variable and f is a function constant. This example also shows that the JP procedure does not generalize Robinson's first-order procedure gracefully. To address this issue, our procedure detects whether a unification problem belongs to a fragment for which unification is decidable and finite complete sets of unifiers (CSUs) exist. We call algorithms that enumerate elements of the CSU for such fragments *oracles*. Noteworthy fragments with oracles are first-order terms, patterns [124], functions-as-constructors [105], and a new fragment presented in Sect. 4.5. The unification procedures of Isabelle and Leo-III check whether the unification problem belongs to a decidable fragment, but we take this idea a step further by checking this more efficiently, and for every subproblem arising during unification.

Second, the JP procedure computes many redundant unifiers. Consider the example $F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}$, where it produces, in addition to the desired unifiers $\{F \mapsto \lambda x.\, H\}$ and $\{G \mapsto \lambda x.\, \mathsf{b}\}$, the redundant unifier $\{F \mapsto \lambda x.\, H,\ G \mapsto \lambda x.\, x\}$. The design of our procedure avoids computing many redundant unifiers, including this one. Additionally, as oracles usually return a small CSU, their integration reduces the number of redundant unifiers.

Third, the JP procedure applies more explosive rules than Huet's preunification procedure to flex-rigid pairs. To gracefully generalize Huet's procedure, we show that his rules for flex-rigid pairs suffice to enumerate CSUs if combined with appropriate rules for flex-flex pairs.

Fourth, the JP procedure repeatedly traverses the parts of the unification problem that have already been unified. Consider the problem $\mathsf{f}^{100}(G\,\mathsf{a}) \stackrel{?}{=} \mathsf{f}^{100}(H\,\mathsf{b})$, where the exponents denote repeated application. It is easy to see that this problem can be reduced to $G\,\mathsf{a} \stackrel{?}{=} H\,\mathsf{b}$. However, the JP procedure will wastefully retraverse the common context $\mathsf{f}^{100}[\,]$ after applying each new substitution. Since the JP procedure must apply substitutions to the variables occurring in the common context above the position of disagreement, it cannot be easily adapted to eagerly decompose unification pairs. By contrast, our procedure is designed to decompose the pairs eagerly, never traversing a common context twice.

Last, the JP procedure does not allow applying substitutions and β-reducing lazily. The rules of simpler procedures (e.g., first-order [79] and pattern unification [124]) depend on only the heads of the unification pair. Thus, to determine the next step, implementations of these procedures need to substitute and β-reduce until only the heads of the current unification pair are not mapped by the substitution and are not λ-abstractions. Since the JP procedure is not based on the decomposition of unification pairs, it is unfit for optimizations of this kind. We designed our procedure to allow for this optimization.

To more efficiently find terms (in a large term set) that are unifiable with a given query term, we developed a higher-order extension of fingerprint indexing [144] (Sect. 4.6). We implemented our procedure, several oracles, and the fingerprint index (Sect. 4.7) in the Zipperposition prover [48, 49]. Since a straightforward implementation of the JP procedure already existed in Zipperposition, we used it as a baseline to evaluate the performance of our procedure (Sect. 4.8). The results show substantial performance improvements.

4.2 Background

Our setting is the simply typed λ-calculus. Unless mentioned otherwise, we use the same notation as laid out in Chapter 2. Additional notions are introduced as follows.

Parameters and *body* for any term $\lambda\overline{x}.\,s$ are defined to be $\overline{x}$ and s respectively, where s is not a λ-abstraction. The *size* of a term is inductively defined as $\mathrm{size}(F) = 1$; $\mathrm{size}(x) = 1$; $\mathrm{size}(\mathsf{f}) = 1$; $\mathrm{size}(s\,t) = \mathrm{size}(s) + \mathrm{size}(t)$; $\mathrm{size}(\lambda x.\,s) = \mathrm{size}(s) + 1$. A term is in *head normal form* (*hnf*) if it is of the form $\lambda\overline{x}.\,a\,\overline{t}$, where a is a free variable, a bound variable, or a constant. In this case, a is called the *head* of the term. Note that this relaxes the condition that the term needs to be in β-normal form to determine its head. A term is called flex or rigid if its head is flex or rigid, respectively. By $s\downarrow_{\mathsf{h}}$ we denote the term obtained from a term s by repeated β-reduction of the leftmost outermost redex until it is in hnf. Unless stated otherwise, we view terms syntactically, as opposed to $\alpha\beta\eta$-equivalence classes. The common context $\mathscr{C}(s,t)$ of two η-long β-reduced terms s and t of the same type is defined inductively as follows, assuming that $a \neq b$: $\mathscr{C}(\lambda x.\,s, \lambda y.\,t) = \lambda x.\,\mathscr{C}(s, \{y \mapsto x\}t)$; $\mathscr{C}(a\,\overline{s}_m, b\,\overline{t}_n) = \Box$; $\mathscr{C}(a\,\overline{s}_m, a\,\overline{t}_m) = a\,\mathscr{C}(s_1, t_1)\ldots\mathscr{C}(s_m, t_m)$. Unless otherwise stated, we take the unification constraint $s \stackrel{?}{=} t$ to be an unordered pair of two terms of the same type. To ease notation, we do not write parentheses around application of substitutions to terms (or other objects containing terms); in other words we shorten $\sigma(\theta(\varrho(s)))$ to $\sigma\theta\varrho\,s$. When we write $\tau_1 \to \cdots \to \tau_n \to \upsilon$ we assume υ to be a base type, by convention.

Remark We use the definition of a CSU from Sect. 2.5.2 because JP's definition of a CSU, which we have adopted in our earlier work [168], is flawed. JP's definition does not employ the notion of auxiliary variables, but instead requires $\varrho X = \theta\sigma X$ for all variables mapped by ϱ. This is problematic because nothing prevents ϱ from mapping the auxiliary variables. For example, $\sigma = \{F \mapsto \lambda xy.\, G\, y\}$ is supposed to be an MGU for $F\,\mathsf{a}\,\mathsf{c} \stackrel{?}{=} F\,\mathsf{b}\,\mathsf{c}$. But for the unifier $\varrho = \{F \mapsto \lambda xy.\, y,\, G \mapsto \lambda x.\, \mathsf{d}\}$, without the notion of auxiliary variables, there exists no appropriate substitution θ because $\varrho G = \theta\sigma G$ requires $\theta G = \lambda x.\, \mathsf{d}$ and $\varrho F = \theta\sigma F$ requires $\theta G = \lambda x.\, x$. By declaring G as an auxiliary variable, we focus only on the important variables from the initial problem (in this case only F), which allows σ to be an MGU.

4.3 The Unification Procedure

To unify two terms s and t, our procedure builds a tree as follows. The nodes of the tree have the form (E, σ), where E is a multiset of unification constraints $\{(s_1 \stackrel{?}{=} t_1), \ldots, (s_n \stackrel{?}{=} t_n)\}$ and σ is the substitution constructed up to that point. The root node is $(\{s \stackrel{?}{=} t\}, \mathrm{id})$, where id is the identity substitution. The tree is then constructed applying the transitions listed below. The leaves of the tree are either failure nodes $\bot$ or substitutions σ. Ignoring failure nodes, the set of all substitutions in the leaves forms a complete set of unifiers for s and t. More generally, our procedure can be used to unify a multiset E of constraints by making the root of the unification tree (E, id).

The procedure requires an infinite supply of fresh free variables that must be disjoint from the variables occurring in the initial multiset E. Whenever a transition $(E, \sigma) \longrightarrow (E', \sigma')$ is made, all fresh variables used in σ' are removed from the supply and cannot be used again as fresh variables.

The transitions are parameterized by a mapping $\mathscr{P}$ that assigns a set of substitutions to a unification pair; this mapping abstracts the concept of unification rules present in other unification procedures. Moreover, the transitions are parameterized by a selection function S mapping a multiset E of unification constraints to one of those constraints $S(E) \in E$, the *selected* constraint in E. In this chapter, we consider *freshness* of the variables with respect to the free variables occurring in E. The transitions, defined as follows, are only applied if the grayed constraint is selected.

Succeed $\quad (\emptyset, \sigma) \longrightarrow \sigma$

Normalize$_{\alpha\eta}$ $\quad (\{\lambda\overline{x}_m.\, s \stackrel{?}{=} \lambda\overline{y}_n.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{z}_m.\, s' \stackrel{?}{=} \lambda\overline{z}_m.\, t'\, z_{n+1} \ldots z_m\} \uplus E, \sigma)$
where $m \geq n$, $\overline{x}_m \neq \overline{y}_n$, s and t are not λ-abstractions, $\overline{z}_m$ are bound variables fresh with respect to s and t, $s' = \{x_1 \mapsto z_1, \ldots, x_m \mapsto z_m\} s$, and $t' = \{y_1 \mapsto z_1, \ldots, y_n \mapsto z_n\} t$

Normalize$_{\beta}$ $\quad (\{\lambda\overline{x}.\, s \stackrel{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}.\, s \downarrow_{\mathsf{h}} \stackrel{?}{=} \lambda\overline{x}.\, t \downarrow_{\mathsf{h}}\} \uplus E, \sigma)$
where s or t is not in hnf

Dereference $\quad (\{\lambda\overline{x}.\, F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}.\, (\sigma F)\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma)$
where none of the previous transitions applies and F is mapped by σ

Fail $\quad (\{\lambda\overline{x}.\, a\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, b\,\overline{t}_n\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions applies, and a and b are different rigid heads

Delete $(\{s \stackrel{?}{=} s\} \uplus E, \sigma) \longrightarrow (E, \sigma)$
where none of the previous transitions applies

OracleSucc $(\{s \stackrel{?}{=} t\} \uplus E, \sigma) \longrightarrow (E, \varrho\sigma)$
where none of the previous transitions applies, some oracle found a finite CSU U for $\sigma s \stackrel{?}{=} \sigma t$ using fresh auxiliary variables, and $\varrho \in U$; if multiple oracles found a CSU, only one of them is considered

OracleFail $(\{s \stackrel{?}{=} t\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions applies, and some oracle determined $\sigma s \stackrel{?}{=} \sigma t$ has no solutions

Decompose $(\{\lambda\overline{x}.\, a\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, a\,\overline{t}_m\} \uplus E, \sigma) \longrightarrow (\{s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\} \uplus E, \sigma)$
where none of the transitions Succeed to OracleFail applies

Bind $(\{s \stackrel{?}{=} t\} \uplus E, \sigma) \longrightarrow (\{s \stackrel{?}{=} t\} \uplus E, \varrho\sigma)$
where none of the transitions Succeed to OracleFail applies, and $\varrho \in \mathscr{P}(s \stackrel{?}{=} t)$.

The transitions are designed so that only OracleSucc, Decompose, and Bind can introduce parallel branches in the constructed tree. OracleSucc can introduce branches using different unifiers of the CSU, Bind can introduce branches using different substitutions in $\mathscr{P}$, and Decompose can be applied in parallel with Bind.

The forms of the rules OracleSucc and Bind are similar: both extend the current substitution. However, they are designed following different principles. OracleSucc solves the selected unification constraint using an efficient algorithm applicable only to certain classes of terms. On the other hand, Bind is applied to explore the whole search space for any given constraint. To compute a CSU using efficient oracles and to discover failures early, Bind is applicable only if OracleSucc (or OracleFail) is not.

Our approach is to apply substitutions and $\alpha\beta\eta$-normalize terms lazily. In this context, laziness means that the transitions $\mathsf{Normalize}_{\alpha\eta}$, $\mathsf{Normalize}_{\beta}$, and Dereference partially normalize and partially apply the constructed substitution just enough to ensure that the heads are the ones that would be obtained if the substitution was fully applied and the term was fully normalized. Additionally, the transitions that modify the constructed substitution, OracleSucc and Bind, do not apply that substitution to the unification pairs directly, but only extend it with a new binding. To support lazy dereferencing, these rules must maintain the invariant that all substitutions are idempotent. The invariant is easily preserved if the substitution ϱ from the definitions of OracleSucc and Bind is itself idempotent and no variable mapped by σ occurs in ϱF, for any variable F mapped by ϱ.

The OracleSucc and OracleFail transitions invoke oracles, such as pattern unification, to compute a CSU faster, produce fewer redundant unifiers, and discover nonunifiability earlier. In some cases, the addition of oracles lets the procedure terminate more often.

In the literature, oracles are usually stated under the assumption that their input belongs to the appropriate fragment. To check whether a unification constraint is inside the fragment, the substitution must be fully applied and the constraint must be β-normalized. To avoid these expensive operations and enable efficient oracle integration, oracles must be redesigned to lazily discover whether the terms belong to their fragment. Most oracles

contain a decomposition operation that requires only a partial application of the substitution and only partial β-normalization. If one of the constraints resulting from decomposition is not in the fragment, the original problem is not in the fragment. This allows detecting that the problem is not in the fragment without fully applying the substitution and β-normalizing.

The core of the procedure lies in the Bind step, parameterized by the mapping $\mathscr{P}$ that determines which substitutions (*bindings*) to create. The bindings are defined as follows:

JP-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some α_i is equal to β and $n > 0$. Then the JP-style projection binding is

$$F \mapsto \lambda \overline{x}_n.\, x_i$$

Huet-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some $\alpha_i = \gamma_1 \to \cdots \to \gamma_m \to \beta$, $n > 0$ and $m \geq 0$. Huet-style projection is

$$F \mapsto \lambda \overline{x}_n.\, x_i\,(F_1\,\overline{x}_n)\,\ldots\,(F_m\,\overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Imitation of a **for** F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and a be a free variable or a constant of type $\gamma_1 \to \cdots \to \gamma_m \to \beta$ where $n, m \geq 0$. The imitation binding is

$$F \mapsto \lambda \overline{x}_n.\, a\,(F_1\,\overline{x}_n)\ldots(F_m\,\overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Elimination for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where $n > 0$. In addition, let $1 \leq j_1 < \cdots < j_i \leq n$ and $i < n$. Elimination for the sequence $(j_k)_{k=1}^{i}$ is

$$F \mapsto \lambda \overline{x}_n.\, G\,x_{j_1}\,\ldots\,x_{j_i}$$

where the fresh free variable G as well as all x_{j_k} are of appropriate types. We call fresh variables emerging from this binding in the role of G *elimination variables*.

Identification for F **and** G Let F and G be different free variables. Furthermore, let the type of F be $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and the type of G be $\gamma_1 \to \cdots \to \gamma_m \to \beta$, where $n, m \geq 0$. Then, the identification binding binds F and G with

$$F \mapsto \lambda \overline{x}_n.\, H\,\overline{x}_n\,(F_1\,\overline{x}_n)\ldots(F_m\,\overline{x}_n) \quad G \mapsto \lambda \overline{y}_m.\, H\,(G_1\,\overline{y}_m)\ldots(G_n\,\overline{y}_m)\,\overline{y}_m$$

where the fresh free variables $H, \overline{F}_m, \overline{G}_n$ and bound variables $\overline{x}_n, \overline{y}_m$ are of appropriate types. Fresh variables from this binding with the role of H are called *identification variables*.

Iteration for F Let F be a free variable of the type $\alpha_1 \to \cdots \to \alpha_n \to \beta_1$ and let some α_i be the type $\gamma_1 \to \cdots \to \gamma_m \to \beta_2$, where $n > 0$ and $m \geq 0$. Iteration for F at i is

$$F \mapsto \lambda \overline{x}_n.\, H\,\overline{x}_n\,(\lambda \overline{y}.\, x_i\,(G_1\,\overline{x}_n\,\overline{y})\ldots(G_m\,\overline{x}_n\,\overline{y}))$$

The free variables H and $G_1, \ldots, G_m$ are fresh, and $\overline{y}$ is an arbitrary-length sequence of bound variables of arbitrary types. All new variables are of appropriate types. Due to indeterminacy of $\overline{y}$, this step is infinitely branching.

The following mapping $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, s \stackrel{?}{=} \lambda\overline{x}.\, t)$ is used as the parameter $\mathscr{P}$ of the procedure. It provides all the bindings that the procedures needs to be complete, and it is defined as follows:

- If the constraint is rigid-rigid, $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, s \stackrel{?}{=} \lambda\overline{x}.\, t) = \emptyset$.
- If the constraint is flex-rigid, let $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, a\,\overline{t})$ be
  - an imitation of a for F, if a is a constant, and
  - all Huet-style projections for F, if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, G\,\overline{t})$ be
  - all identifications and iterations for both F and G, and
  - all JP-style projections for non-identification variables among F and G.
- If the constraint is flex-flex and the heads are identical, we distinguish two cases:
  - if the head is an elimination variable, $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, s \stackrel{?}{=} \lambda\overline{x}.\, t) = \emptyset$;
  - otherwise, let $\mathscr{P}_{\mathsf{c}}(\lambda\overline{x}.\, F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{t})$ be all iterations for F at arguments of functional type and all eliminations for F.

Comparison with the JP Procedure The JP procedure enumerates unifiers by constructing a search tree with nodes of the form $(s \stackrel{?}{=} t, \sigma)$, where $s \stackrel{?}{=} t$ is the current unification problem and σ is the substitution built so far. The initial node consists of the input problem and the identity substitution. Success nodes are nodes of the form $(s \stackrel{?}{=} s, \sigma)$. The set of all substitutions contained in the success nodes form a CSU.

To determine the child nodes of a node $(s \stackrel{?}{=} t, \sigma)$, the procedure computes the common context C of s and t, yielding term pairs $(s_1, t_1), \ldots, (s_n, t_n)$, called *disagreement pairs*, such that $s = C[s_1, \ldots, s_n]$ and $t = C[t_1, \ldots, t_n]$. It chooses one of the disagreement pairs (s_i, t_i). Depending on the context C and the chosen disagreement pair (s_i, t_i), it determines a set of bindings $\mathscr{P}_{\mathsf{JP}}(C, s_i, t_i)$. For each of the bindings $\varrho \in \mathscr{P}_{\mathsf{JP}}(C, s_i, t_i)$, it creates a child node $((\varrho s)\downarrow_{\beta\eta} \stackrel{?}{=} (\varrho t)\downarrow_{\beta\eta}, \varrho\sigma)$, where $u\downarrow_{\beta\eta}$ denotes a $\beta\eta$-normal form of a term u.

The set of bindings $\mathscr{P}_{\mathsf{JP}}(C, s_i, t_i)$ is based on the heads of s_i and t_i, and the free variables occurring above s_i and t_i in C. The set $\mathscr{P}_{\mathsf{JP}}(C, s_i, t_i)$ contains

- all JP-style projections for free variables that are heads of s_i or t_i;[1]
- an imitation of a for F if a free variable F is the head of s_i and a free variable or constant a is the head of t_i (or vice versa);
- all eliminations for free variables occurring above the chosen disagreement pair eliminating only the argument containing the disagreement pair;
- an identification for the heads of s_i and t_i if they are both free variables; and

[1] In JP's formulation of projection, they explicitly mention that the projected argument must be of base type. In our presentation, this follows from β being of base type by the convention introduced in Sect. 2.3.

- all iterations for the heads of s_i and t_i if they are free variables, and for all free variables occurring above the disagreement pair.[2]

Architecturally, the most noticeable difference between the JP procedure and ours is the representation of the problem: The JP procedure works on a single constraint, while our procedure maintains a multiset of constraints. At a first glance, this is a merely presentational change. However, it has consequences for termination, performance, and redundancy of the procedure.

Since the JP procedure never decomposes the common context of its only constraint, it allows iteration or elimination to be applied at a free variable above the disagreement pair, even if bindings were already applied below that free variable. This can lead to many different paths to the same unifier. In contrast, our procedure decides which binding to apply to a flex-flex pair with the same head as soon as it is observed. Also, it explores the possibility of not applying a binding and decomposing the pair. In either case, the flex-flex pair is never revisited, which improves the performance and returns fewer redundant unifiers. We show that this restriction prunes the search space without losing completeness.

Our procedure chooses child nodes based on only the heads of the selected unification constraint. In contrast, the JP procedure tracks all the variables occurring in the common context. Thus, lazy normalization and lazy variable substitution cannot be integrated in the JP procedure a straightforward fashion. Moreover, as the JP procedure does not feature a rule similar to Decompose, it always retraverses the already unified part of the problem, resulting in poor performance on deep terms.

One of the main drawbacks of the JP procedure is that it features a highly explosive, infinitely branching iteration rule. This rule is a more general version of Huet-style projection. Its universality enables finding elements of the CSU for flex-flex pairs, for which Huet-style projection does not suffice. However, the JP procedure applies iteration indiscriminately on both flex-flex and flex-rigid pairs. We discovered that our procedure remains complete if iteration is applied only on flex-flex pairs, and Huet-style projection only on flex-rigid ones. This helps our procedure terminate more often than the JP procedure. As a side-effect, the restriction of our procedure to the preunification problem is a graceful generalization of Huet procedure, with additional improvements such as oracles, lazy substitution, and lazy β-reduction.

The bindings of our procedure contain further optimizations that are absent from the JP procedure: The JP procedure applies eliminations for one parameter at a time, yielding multiple paths to the same unifier. It applies imitations to flex-flex pairs, which we found to be unnecessary. Similarly, we found out that tracking which rules introduced which variables can avoid computing redundant unifiers: It is not necessary to apply iterations and eliminations on elimination variables, and projections on identification variables.

Examples We present some examples that demonstrate advantages of our procedure. The displayed branches of the constructed trees are not necessarily exhaustive. We abbreviate JP-style projection as JP Proj, imitation as Imit, identification as Id, Decompose as Dc, Dereference as Dr, Normalize$_\beta$ as N$_\beta$, and Bind of a binding b as B(b). Transitions of the JP

[2] In JP's formulation of iteration, it is not immediately obvious whether they intend to require iteration of arguments of base type. However, their Definition 2.4 [86] shows that they do.

procedure are denoted by $\Longrightarrow$. For the JP transitions we implicitly apply the generated bindings and fully normalize terms, which significantly shortens JP derivations.

Example 4.1. The JP procedure does not terminate on the problem $G \stackrel{?}{=} \mathsf{f}\,G$:

$$(G \stackrel{?}{=} \mathsf{f}\,G, \mathrm{id}) \overset{\text{Imit}}{\Longrightarrow} (\mathsf{f}\,G' \stackrel{?}{=} \mathsf{f}^2\,G', \sigma_1) \overset{\text{Imit}}{\Longrightarrow} (\mathsf{f}^2\,G'' \stackrel{?}{=} \mathsf{f}^3\,G'', \sigma_2) \overset{\text{Imit}}{\Longrightarrow} \cdots$$

where $\sigma_1 = \{G \mapsto \lambda x.\mathsf{f}\,G'\}$ and $\sigma_2 = \{G' \mapsto \lambda x.\mathsf{f}\,G''\}\sigma_1$. By including any oracle that supports the first-order occurs check, such as the pattern oracle or the fixpoint oracle described in Sect. 4.7, our procedure gracefully generalizes first-order unification:

$$(\{G \stackrel{?}{=} \mathsf{f}\,G\}, \mathrm{id}) \overset{\text{OracleFail}}{\longrightarrow} \bot$$

Example 4.2. The following derivation illustrates the advantage of the Decompose rule.

$$\begin{aligned}
&(\{\mathsf{h}^{100}\,(F\,\mathsf{a}) \stackrel{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b})\}, \mathrm{id}) \overset{\text{Dc}^{100}}{\longrightarrow} (\{F\,\mathsf{a} \stackrel{?}{=} G\,\mathsf{b}\}, \mathrm{id}) \overset{\text{B(Id)}}{\longrightarrow} (\{F\,\mathsf{a} \stackrel{?}{=} G\,\mathsf{b}\}, \sigma_1) \\
&\overset{\text{Dr+N}_\beta}{\longrightarrow} (\{H\,\mathsf{a}\,(F'\,\mathsf{a}) \stackrel{?}{=} H\,(G'\,\mathsf{b})\,\mathsf{b}\}, \sigma_1) \overset{\text{Dc}}{\longrightarrow} (\{\mathsf{a} \stackrel{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \sigma_1) \\
&\overset{\text{B(Imit)}}{\longrightarrow} (\{\mathsf{a} \stackrel{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \sigma_2) \overset{\text{Dr+N}_\beta}{\longrightarrow} (\{\mathsf{a} \stackrel{?}{=} \mathsf{a}, F'\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \sigma_2) \overset{\text{Delete}}{\longrightarrow} (\{F'\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \sigma_2) \\
&\overset{\text{B(Imit)}}{\longrightarrow} (\{F'\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \sigma_3) \overset{\text{Dr+N}_\beta}{\longrightarrow} (\{\mathsf{b} \stackrel{?}{=} \mathsf{b}\}, \sigma_3) \overset{\text{Delete}}{\longrightarrow} (\emptyset, \sigma_3) \overset{\text{Succeed}}{\longrightarrow} \sigma_3
\end{aligned}$$

where $\sigma_1 = \{F \mapsto \lambda x.H\,x\,(F'\,x), G \mapsto \lambda y.H\,(G'\,y)\,y\}$; $\sigma_2 = \{G' \mapsto \lambda x.\mathsf{a}\}\sigma_1$; and $\sigma_3 = \{F' \mapsto \lambda x.\mathsf{b}\}\sigma_2$. The JP procedure produces the same intermediate substitutions σ_1 to σ_3, but since it does not decompose the terms, it retraverses the common context $\mathsf{h}^{100}\,[\,]$ at every step to identify the contained disagreement pair:

$$\begin{aligned}
&(\mathsf{h}^{100}\,(F\,\mathsf{a}) \stackrel{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b}), \mathrm{id}) \overset{\text{Id}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \stackrel{?}{=} \mathsf{h}^{100}\,(H\,(G'\,\mathsf{b})\,\mathsf{b}), \sigma_1) \\
&\overset{\text{Imit}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \stackrel{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_2) \overset{\text{Imit}}{\Longrightarrow} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}) \stackrel{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_3) \overset{\text{Succeed}}{\Longrightarrow} \sigma_3
\end{aligned}$$

Example 4.3. Even when no oracles are used, our procedure performs better than the JP procedure on small, simple problems. Consider the problem $F\,\mathsf{a} \stackrel{?}{=} \mathsf{a}$, which has a two element CSU: $\{F \mapsto \lambda x.x, F \mapsto \lambda x.\mathsf{a}\}$. Our procedure terminates, finding both unifiers:

$$\begin{aligned}
&(\{F\,\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \mathrm{id}) \overset{\text{B(JP Proj)}}{\longrightarrow} (\{F\,\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \{F \mapsto \lambda x.x\}) \overset{\text{Dr+N}_\beta}{\longrightarrow} (\{\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \{F \mapsto \lambda x.x\}) \\
&\overset{\text{Delete}}{\longrightarrow} (\emptyset, \{F \mapsto \lambda x.x\}) \overset{\text{Succeed}}{\longrightarrow} \{F \mapsto \lambda x.x\} \\
&(\{F\,\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \mathrm{id}) \overset{\text{B(Imit)}}{\longrightarrow} (\{F\,\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \{F \mapsto \lambda x.\mathsf{a}\}) \overset{\text{Dr+N}_\beta}{\longrightarrow} (\{\mathsf{a} \stackrel{?}{=} \mathsf{a}\}, \{F \mapsto \lambda x.\mathsf{a}\}) \\
&\overset{\text{Delete}}{\longrightarrow} (\emptyset, \{F \mapsto \lambda x.\mathsf{a}\}) \overset{\text{Succeed}}{\longrightarrow} \{F \mapsto \lambda x.\mathsf{a}\}
\end{aligned}$$

The JP procedure finds those two unifiers as well, but it does not terminate as it applies iterations to F.

Example 4.4. The search space restrictions also allow us to prune some redundant unifiers. Consider the problem $F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}$, where a and b are of base type. Our procedure produces only one failing branch and the following two successful branches:

$$(\{F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{Dc}} (\{G\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Imit)}} (\{G\,\mathsf{a} \stackrel{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\,\mathsf{b}\})$$
$$\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{b} \stackrel{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\,\mathsf{b}\}) \xrightarrow{\mathsf{Delete}} (\emptyset, \{G \mapsto \lambda x.\,\mathsf{b}\}) \xrightarrow{\mathsf{Succeed}} \{G \mapsto \lambda x.\,\mathsf{b}\}$$

$$(\{F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Elim)}} (\{F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}\}, \{F \mapsto \lambda x.\,F'\})$$
$$\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{F' \stackrel{?}{=} F'\}, \{F \mapsto \lambda x.\,F'\}) \xrightarrow{\mathsf{Delete}} (\emptyset, \{F \mapsto \lambda x.\,F'\}) \xrightarrow{\mathsf{Succeed}} \{F \mapsto \lambda x.\,F'\}$$

The JP procedure additionally produces the following redundant unifier:

$$(F(G\,\mathsf{a}) \stackrel{?}{=} F\,\mathsf{b}, \mathrm{id}) \xRightarrow{\mathsf{JP\ Proj}} (F\,\mathsf{a} = F\,\mathsf{b}, \{G \mapsto \lambda x.\,x\})$$
$$\xRightarrow{\mathsf{Elim}} (F' = F', \{G \mapsto \lambda x.\,x, F \mapsto \lambda x.\,F'\}) \xRightarrow{\mathsf{Succeed}} \{G \mapsto \lambda x.\,x, F \mapsto \lambda x.\,F'\}$$

Moreover, the JP procedure does not terminate because an infinite number of iterations is applicable at the root. Our procedure terminates in this case since we only apply iteration binding for non base-type arguments, which F does not have.

Pragmatic Variant We structured our procedure so that most of the unification machinery is contained in the Bind step. Modifying $\mathscr{P}$, we can sacrifice completeness and obtain a pragmatic variant of the procedure that often performs better in practice. Our preliminary experiments showed that using the mapping $\mathscr{P}_\mathsf{p}$ defined as follows gives good performance, while finding most useful unifiers:

- If the constraint is rigid-rigid, $\mathscr{P}_\mathsf{p}(\lambda\overline{x}.\,s \stackrel{?}{=} \lambda\overline{x}.\,t) = \emptyset$.
- If the constraint is flex-rigid, let $\mathscr{P}_\mathsf{p}(\lambda\overline{x}.\,F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{t})$ be
  - an imitation of a for F, if a is a constant, and
  - all Huet-style projections for F if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathscr{P}_\mathsf{p}(\lambda\overline{x}.\,F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\,G\,\overline{t})$ be
  - an identification binding for F and G, and
  - all Huet-style projections for F if F is not an identification variable
- If the constraint is flex-flex and the heads are identical, we distinguish two cases:
  - if the head is an elimination variable, $\mathscr{P}_\mathsf{p}(\lambda\overline{x}.\,F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\,F\,\overline{t}) = \emptyset$;
  - otherwise, let $\mathscr{P}_\mathsf{p}(\lambda\overline{x}.\,F\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\,F\,\overline{t})$ be the set of all eliminations bindings for F.

The pragmatic variant of our procedure removes all iteration bindings to enforce finite branching. In addition, it imposes limits on the number of bindings applied, counting the applications of bindings locally, per constraint, as follows. It is useful to distinguish the Huet-style projection cases where α_i is a base type (called *simple projection*), which always reduces the problem size, and the cases where α_i is a functional type (called *functional projection*). We limit the number of applications of the following bindings: functional projections, eliminations, imitations, and identifications. In addition, a limit on the total number of applied bindings can be set. An elimination binding that removes k arguments counts as k elimination steps. Thanks to these limits, the pragmatic variant terminates.

To fail as soon as any of the limits is reached, the pragmatic variant employs an additional oracle. If this oracle determines that the limits are reached and the constraint is of the form $\lambda\overline{x}.F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.G\,\overline{t}_n$, it returns a *trivial unifier* – a substitution $\{F \mapsto \lambda\overline{x}_m.H, G \mapsto \lambda\overline{x}_n.H\}$, where H is a fresh variable; if a limit is reached and the constraint is flex-rigid, the oracle fails; if the limits are not reached, it reports that terms are outside its fragment. The trivial unifier prevents the procedure from failing on easily unifiable flex-flex pairs.

Careful tuning of each limit optimizes the procedure for a specific class of problems. For problems originating from proof assistants, a low unification depth usually suffices. However, hard hand-crafted problems often need deeper unification.

4.4 Proof of Completeness

Like the JP procedure, our procedure, parameterized with $\mathscr{P}_\mathsf{c}$, misses no unifiers:

Theorem 4.5. *The procedure described in Sect. 4.3 parameterized by $\mathscr{P}_\mathsf{c}$ is complete, meaning that the substitutions on the leaves of the constructed tree form a CSU. More precisely, let E be a multiset of constraints and let V be the supply of fresh variables provided to the procedure. Then for any unifier ϱ of E there exists a derivation $(E, \mathrm{id}) \longrightarrow^* \sigma$ and a substitution θ such that for all free variables $X \notin V$, we have $\varrho X = \theta\sigma X$.*

Taking a high-level view, this theorem is proved by incrementally defining states (E_j, σ_j) and *remainder substitutions* ϱ_j starting with $(E_0, \sigma_0) = (E, \mathrm{id})$ and $\varrho_0 = \varrho$. The substitution ϱ_j is what remains to be added to σ_j to reach ϱ_0. States are defined so that the shape of the selected constraint from E_j and the remainder substitution guide the choice of applicable transition rule. Finally, a measure based on values of E_j and ϱ_j that decreases with each application of the rules is employed. Therefore, eventually, the target substitution σ will be reached.

In the remainder of this section, we view terms as $\alpha\beta\eta$-equivalence classes, with the η-long β-normal form as their canonical representative. Moreover, we consider all substitutions to be fully applied. These assumptions are justified because all bindings depend only on the heads of terms and hence replacing the lazy transitions $\mathsf{Normalize}_{\alpha\eta}$, $\mathsf{Normalize}_{\beta}$, and Dereference by eager counterparts only affects the efficiency but not the overall behavior of our procedure.

We now give the detailed completeness proof of Theorem 4.5. Our proof is an adaptation of the proof given by Jensen and Pietrzykowski [86]. Definitions and lemmas are reused, but are combined together differently to suit our procedure. We start by listing all reused definitions and lemmas from the original JP proof. The "JP" labels in their statements refer to the corresponding lemmas and definitions from the original proof.

Definition 4.6 (JP D1.6)**.** Given two terms t and s and their common context C, we can write t as $C[\overline{t}]$ and s as $C[\overline{s}]$ for some $\overline{t}$ and $\overline{s}$. The pairs (s_j, t_j) are called *disagreement pairs.*

Definition 4.7 (JP D3.1)**.** Given two terms t and s, let $\lambda\overline{x}.\, t'$ and $\lambda\overline{y}.\, s'$ be respective α-equivalent terms such that their parameters $\overline{x}$ and $\overline{y}$ are disjoint. Then the disagreement pairs of t' and s' are called *opponent pairs* in t and s.

Example 4.8. To determine the opponent pairs of $\lambda x.\, \mathsf{g}(\mathsf{g}x)$ and $\lambda x.\, \mathsf{g}\, x$, we first α-rename the second term to $\lambda y.\, \mathsf{g}\, y$. Then, we identify the common context of $\mathsf{g}(\mathsf{g}x)$ and $\mathsf{g}\, y$ as $\mathsf{g}\,\square$, which leads us to the opponent pair $(\mathsf{g}\, x, y)$.

Lemma 4.9 (JP L3.3 (1))**.** *Let ϱ be a substitution and X, Y be free variables such that $\varrho(X\,\overline{s}) = \varrho(Y\,\overline{t})$ for some term tuples $\overline{s}$ and $\overline{t}$. Then for every opponent pair u, v in ϱX and ϱY (Definition 4.7), the head of u or v is a parameter of ϱX or ϱY.*

Example 4.10. Let ϱ be the substitution $\{X \mapsto \lambda x.\, \mathsf{f}\, x, Y \mapsto \lambda x.\, x\}$. Clearly, ϱ unifies the problem $X\,\mathsf{a} \stackrel{?}{=} Y\,(\mathsf{f}\,\mathsf{a})$. The opponent pair for $\varrho\, X$ and $\varrho\, Y$ is $(\mathsf{f}\, x, y)$, which clearly adheres to the conclusion of Lemma 4.9. Intuitively, one of the consequences of this lemma is that any solution for flex-flex problems that does not mention parameters will not have opponent pairs. For example, one solution for the problem $X\,\mathsf{a} \stackrel{?}{=} Y\,(\mathsf{f}\,\mathsf{a})$ is $\varrho = \{X \mapsto \lambda x.\, \mathsf{c}, Y \mapsto \lambda x.\, \mathsf{c}\}$, which has no opponent pairs.

In contrast to applied constants, applied variables should not be eagerly decomposed. For a constant f, if $\mathsf{f}\,\overline{s} \stackrel{?}{=} \mathsf{f}\,\overline{t}$ has a unifier, that unifier must clearly also unify $s_i \stackrel{?}{=} t_i$ for each i. For a free variable X, a unifier of $X\,\overline{s} \stackrel{?}{=} X\,\overline{t}$ does not necessarily unify $s_i \stackrel{?}{=} t_i$. For example, $\{X \mapsto \lambda\overline{x}.\, \mathsf{a}\}$ is a unifier that does refer to any of the arguments of X. The concept of ω-simplicity is a criterion on unifiers that captures some of the cases where eager decomposition is possible. Non-ω-simplicity on the other hand is the main trigger of Iteration—the most explosive binding of our procedure.

Definition 4.11 (JP D3.2)**.** An occurrence of a parameter x of term t in the body of t is *ω-simple* if both

1. the arguments of x are distinct and are exactly (the η-long forms of) all of the variables bound in the body of t, and

2. this occurrence of x is not in an argument of any parameter of t.

Example 4.12 (JP E1)**.** Term $\lambda xy.\, \mathsf{f}\,\boldsymbol{x\, y}$ has two and term $\lambda x.\, \mathsf{g}(\lambda y.\, \boldsymbol{x}\, y)$ has one ω-simple occurrence, denoted in boldface. In the following two examples, boldface occurrences are *not* ω-simple: $\lambda x.\, \mathsf{g}(\lambda y.\, \mathsf{f}\, y\, \boldsymbol{x})$, $\lambda xy.\, \mathsf{h}(x\, \boldsymbol{y})$. In the first case x is not applied to parameter y and in the second y occurs as an argument to x.

Definition 4.11 is slightly too restrictive for our purposes. It is unfortunate that condition 1 requires x to be applied to *all* instead of just some of the bound variables. The JP proof would probably work with such a relaxation, and the definition would then cover all cases where eager decomposition is possible. However, to reuse the JP lemmas, we stick to the original notion of ω-simplicity and introduce the following relaxation:

Definition 4.13. An occurrence of a parameter x of term t in the body of t is *base-simple* if it is ω-simple or both

1. x is of base type, and
2. this occurrence of x is not in an argument of any parameter of t.

Lemma 4.14. *Let s have parameters $\overline{x}$ and a subterm $x_j\,\overline{v}$ where this occurrence of x_j is base-simple. Then for any sequence $\overline{t}$ of (at least j) terms, the body of t_j is a subterm of $s\,\overline{t}$ (after normalization) at the position of $x_j\,\overline{v}$ up to renaming of the parameters of t_j. To compare positions of s and $s\,\overline{t}$, we ignore the outermost λ-binders of s.*

Proof. Consider the process of β-normalizing $s\,\overline{t}$. After substituting terms $\overline{t}$ into the body of s, a further reduction can only take place when some t_k is an abstraction that gets arguments in s. The arguments $\overline{v}$ to the x_j are distinct variables bound in the body of s. This follows easily from either case of the definition of base-simplicity. So t_j is applied to the unmodified $\overline{v}$ after substituting terms $\overline{t}$ into the body of s. Base-simplicity also implies that $t_j\,\overline{v}$ does not occur in an argument to another t_k. Hence only the reduction of $t_j\,\overline{v}$ itself affects this subterm. The variables $\overline{v}$ match the parameter count of t_j because we consider the η-long form of t_j; so $t_j\,\overline{v}$ reduces to the body of t_j (modulo renaming). The position is obviously that of $x_j\,\overline{v}$. □

Lemma 4.15 (JP C3.4 strengthened)**.** *Let ϱ be a substitution and X a free variable. If $\varrho(\lambda\overline{x}.X\,\overline{s}) = \varrho\big(\lambda\overline{x}.X\,\overline{t}\big)$ and some occurrence of the ith parameter of ϱX is base-simple, then $\varrho s_i = \varrho t_i$.*

Proof. By Lemma 4.14, ϱs_i occurs in $\varrho X\big(\varrho\overline{s}\big)$ at a certain position that depends only on ϱX. Similarly ϱt_i occurs in $\varrho X\big(\varrho\overline{t}\big) = \varrho X\big(\varrho\overline{s}\big)$ at the same position, and hence $\varrho s_i = \varrho t_i$. □

We define more properties to determine which binding to apply to a given constraint. Roughly speaking, the simple comparison form will trigger identification bindings, projectivity will trigger Huet-style projections, and simple projectivity will trigger JP-style projections.

Definition 4.16 (JP D3.4)**.** We say that s and t are in *simple comparison form* if all ω-simple heads of opponent pairs in s and t are distinct, and each opponent pair has an ω-simple head.

Definition 4.17 (JP D3.5)**.** A term t is called *projective* if the head of t is a parameter of t. If the whole body is just the parameter, then t is called *simply projective.*

A central part of the proof is to find a suitable measure for the remaining problem size. Showing that the measure is strictly decreasing and well founded guarantees that the procedure finds a suitable substitution in finitely many steps. We reuse the measure for remainder substitutions from JP [86], but embed it into a lexicographic measure to handle the decomposition steps and oracles of our procedure.

Definition 4.18 (JP D3.7)**.** The *free weight* of a term t is the total number of occurrences of free variables and constants in t. The *bound weight* of t is the total number of occurrences (excluding occurrences λx) of bound variables in t, but with a particular exemption: if a prefix variable u has one or more ω-simple occurrences in the body, then one such occurrence and its arguments are not counted. It does not matter which occurrence is not counted because in η-long form the bound weight of the arguments of an ω-simple variable is the same for all occurrences of that variable.

Definition 4.19 (JP D3.8)**.** For multisets E of unification constraints and substitutions ϱ, our measure on pairs (E, ϱ) is the lexicographic comparison of

A the sum of the sizes of the terms in ϱE

B the sum over the free weight of ϱF, for all variables F mapped by ϱ

C the sum over the bound weight of ϱF, for all variables F mapped by ϱ

D the sum over the number of parameters of ϱF, for all variables F mapped by ϱ

We denote the quadruple containing these numbers as $\mathrm{ord}(E, \varrho)$. We denote the triple containing only the last three components of $\mathrm{ord}(E, \varrho)$ as $\mathrm{ord}\,\varrho$. We write $<$ for the lexicographic comparison of these triples.

The next six lemmas correspond to the bindings of our procedure and sufficient conditions for the binding to bring us closer to a given solution. This is expressed as a decrease of the ord measure of the remainder. In each of these lemmas, let u be a term with a variable head a and v a term with an arbitrary head b. Let ϱ be a unifier of u and v. The conclusion, let us call it $\boldsymbol{C}$, is always the same: there exists a binding δ applicable to the problem $u \stackrel{?}{=} v$, and there exists a substitution ϱ' such that $\mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho$ and for all variables X except the fresh variables introduced by the binding we have $\varrho X = \varrho'\,\delta X$. For most of these lemmas, we refer to JP [86] for proofs. Although JP only claim $\varrho X = \varrho'\,\delta X$ for variables X mapped by ϱ, inspection of their proofs shows that the equality holds for all X except the fresh variables introduced by the binding. Moreover, some of our bindings have more preconditions, yielding additional orthogonal hypotheses in our lemmas, which we address below.

Lemma 4.20 (JP L3.9)**.** *If $a = b$ is not an elimination variable and ϱa discards any of its parameters, then $\boldsymbol{C}$ by elimination. Moreover, for the elimination variable G introduced by this elimination, $\varrho' G$ discards none of its parameters and has the body of ϱa.*

Proof. Let $\varrho a = \lambda\overline{x}.\,t$ and let $(x_{j_k})_{k=1}^{i}$ be the subsequence of $\overline{x}$ consisting of those variables which occur in the body t. It is a strict subsequence, since ϱa is assumed to discard some parameter. Since the equal heads $a = b$ of the constraint $u \stackrel{?}{=} v$ are not elimination variables, elimination for $(j_k)_{k=1}^{i}$ can be applied. Let $\delta = \{a \mapsto \lambda\overline{x}.\,G\,x_{j_1} \ldots x_{j_i}\}$ be the corresponding binding. Define ϱ' to be like ϱ except

$$\varrho'\,a = a \quad \text{and} \quad \varrho'\,G = \lambda x_{j_1} \ldots x_{j_i}.\,t.$$

Obviously $\varrho' G$ is a closed term and $\varrho X = \varrho'\,\delta X$ holds for all $X \neq G$. Moreover $\mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho$, because free and bound weights stay the same (ϱa and $\varrho' G$ have the same body t) whereas the number of parameters strictly decreases. The definition of $(j_k)_{k=1}^{i}$ implies that $\varrho' G$ discards none of its parameters. □

Lemma 4.21 (JP L3.10)**.** *Assume that there exists a parameter x of ϱa such that x has a non-ω-simple (Definition 4.11) occurrence in ϱa, which is not below another parameter, or such that x has at least two ω-simple occurrences in ϱa. Moreover, if $a = b$, to make iteration applicable, a must not be an elimination variable, and x must be of functional type. Then* **C** *is achieved by iteration.*

Lemma 4.22 (JP L3.11)**.** *Assume that a and b are different free variables. If ϱa is simply projective (Definition 4.17) and a is not an identification variable, then* **C** *by JP-style projection.*

Lemma 4.23 (JP L3.12)**.** *If ϱa is not projective and b is rigid, then* **C** *by imitation.*

Lemma 4.24 (JP L3.13)**.** *Let $a \neq b$. Assume that $\varrho a \neq a$ and $\varrho b \neq b$ are in simple comparison form (Definition 4.16) and neither is projective. Then* **C** *by identification. Moreover, $\varrho' H$ is not projective, where H is the identification variable introduced by this application of the identification binding.*

Proof. This is JP's Lemma 3.13, plus the claim that $\varrho' H$ is not projective. Inspecting the proof of that lemma, it is obvious that $\varrho' H$ cannot be projective because ϱa and ϱb are not projective. □

Lemma 4.25. *Assume that ϱa is projective (Definition 4.17), a is not an identification variable, and b is rigid. Then* **C** *by Huet-style projection.*

Proof. Since ϱa is projective, we have $\varrho a = \lambda \overline{x}_n.\, x_k\, \overline{t}_m$ for some k and some terms $\overline{t}_m$. If ϱa is also simply projective, then x_k must be nonfunctional and since Huet-style projection and JP-style projection coincide in that case, Lemma 4.22 applies. Hence, in the following we may assume that ϱa is not simply projective, i.e., that $m > 0$.

Let δ be the Huet-style projection binding:

$$\delta = \{a \mapsto \lambda \overline{x}_n.\, x_i\, (F_1\, \overline{x}_n)\, \ldots\, (F_m\, \overline{x}_n)\}$$

for fresh variables $F_1, \ldots, F_m$. This binding is applicable because b is rigid. Let ϱ' be the same as ϱ except that we set $\varrho' a = a$ and for each $1 \leq j \leq m$ we set

$$\varrho' F_j = \lambda \overline{x}_n.\, t_j$$

It remains to show that $\operatorname{ord} \varrho' < \operatorname{ord} \varrho$. The free weight of ϱa is the same as the sum of the free weights of $\varrho' F_j$ for $1 \leq j \leq m$. Thus, the free weight is the same for ϱ and ϱ'. The bound weight of ϱa however is exactly 1 larger than the sum of the bound weights of $\varrho' F_j$ for $1 \leq j \leq m$ because of the additional occurrence of x_k in ϱa. The exemption for ω-simple occurrences in the definition of the bound weight cannot be triggered by this occurrence of x_k because $m > 0$ and thus x_k is not ω-simple. It follows that $\operatorname{ord} \varrho' < \operatorname{ord} \varrho$. □

We are now ready to prove the completeness theorem (Theorem 4.5).

Proof. Let E be a multiset of constraints and let V be the supply of fresh variables provided to our procedure. Let ϱ be a unifier of E. We must show that there exist a derivation $(E, \mathrm{id}) \longrightarrow^* \sigma$ and a substitution θ such that for all free variables $X \notin V$, we have $\varrho X = \theta \sigma X$.

Let $E_0 = E$ and $\sigma_0 = \mathrm{id}$. Let $\varrho_0 = \tau \varrho$ for some renaming τ, such that every free variable occurring in $\varrho_0 E_0$ does not occur in E_0 and is not contained in V. Then ϱ_0 unifies E_0

because ϱ unifies E by assumption. Moreover, $\varrho_0 = \varrho_0\sigma_0$. We proceed to inductively define E_j, σ_j and ϱ_j until we reach some j such that $E_j = \emptyset$. To guarantee well-foundedness, we ensure that the measure $\mathrm{ord}(\varrho_j, E_j)$ decreases with each step. We maintain the following invariants for all j:

- $(E_j, \sigma_j) \longrightarrow (E_{j+1}, \sigma_{j+1})$;
- $\mathrm{ord}(E_j, \varrho_j) > \mathrm{ord}(E_{j+1}, \varrho_{j+1})$;
- ϱ_j unifies E_j;
- $\varrho_0 X = \varrho_j\sigma_j X$ for all free variables $X \notin V$;
- every free variable occurring in $\varrho_j E_j$ does not occur in E_j and is not contained in V;
- for every identification variable X, $\varrho_j X$ is not projective; and
- for every elimination variable X, each parameter of $\varrho_j X$ has occurrences in $\varrho_j X$, all of which are base-simple.

If $E_j \neq \emptyset$, let $u \stackrel{?}{=} v$ be the selected constraint $S(E_j)$ in E_j.

First assume that an oracle is able to find a CSU for the constraint $u \stackrel{?}{=} v$. Since ϱ_j unifies u and v, by the definition of a CSU, the CSU discovered by the oracle contains a unifier δ of u and v such that there exists a ϱ_{j+1} and for all free variables X except for the auxiliary variables of the CSU we have $\varrho_j X = \varrho_{j+1}\,\delta X$. Thus, an OracleSucc transition is applicable and yields the node $(E_{j+1}, \sigma_{j+1}) = (\delta(E_j \setminus \{u \stackrel{?}{=} v\}), \delta\,\sigma_j)$. Therefore we have a strict containment $\varrho_{j+1}E_{j+1} \subset \varrho_{j+1}\,\delta\,E_j = \varrho_j E_j$. This implies $\mathrm{ord}(E_{j+1}, \varrho_{j+1}) < \mathrm{ord}(E_j, \varrho_j)$. It also shows that the constraints $\varrho_{j+1}E_{j+1}$ are unified when $\varrho_j E_j$ are. Since the auxiliary variables introduced by OracleSucc are fresh, they can occur neither in E_j nor in $\sigma_j X$ for any $X \notin V$. Hence, we have $\varrho_0 X = \varrho_j\sigma_j X = \varrho_{j+1}\,\delta\,\sigma_j X = \varrho_{j+1}\sigma_{j+1}X$ for all free variables $X \notin V$. Any free variable occurring in $\varrho_{j+1}E_{j+1}$ cannot not occur in E_{j+1} and is not contained in V because $\varrho_{j+1}E_{j+1} \subset \varrho_j E_j$ and the variables in $E_{j+1} = \delta(E_j \setminus \{u \stackrel{?}{=} v\})$ are either variables already present in E_j or fresh variables introduced by OracleSucc. New identification or elimination variables are not introduced; so their properties are preserved. Hence all invariants are preserved.

Otherwise we proceed by a case distinction on the form of $u \stackrel{?}{=} v$. Typically, one of the Lemmas 4.20–4.25 will be applicable. Any one of them gives substitutions ϱ' and δ with properties that let us define $E_{j+1} = \delta E_j$, $\sigma_{j+1} = \delta\,\sigma_j$ and $\varrho_{j+1} = \varrho'$. The problem size always strictly decreases, because these lemmas imply $\varrho_{j+1}E_{j+1} = \varrho_{j+1}\delta E_j = \varrho_j E_j$ and $\mathrm{ord}\,\varrho_{j+1} = \mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho_j$. Regarding the other invariants, the former equation guarantees that ϱ_{j+1} unifies E_{j+1}, and $\varrho_0 X = \varrho_j\,\sigma_j X = \varrho_{j+1}\,\delta\,\sigma_j X = \varrho_{j+1}\,\sigma_{j+1}X$ for all $X \notin V$ because the fresh variables introduced by the binding cannot occur in $\sigma_j X$ for any $X \notin V$. The conditions on variables must be checked separately when new ones are introduced. Let a be the head of $u = \lambda\overline{x}.\,a\,\overline{u}$ and b be the head of $v = \lambda\overline{x}.\,b\,\overline{v}$. Consider the following cases:

u and v have the same head symbol $a = b$

1. Suppose that $\varrho_j a$ has a parameter with a non-base-simple occurrence. By one of the induction invariants, a is not an elimination variable. Among all

non-base-simple occurrences of parameters in $\varrho_j a$, choose the leftmost one, which we call x. This occurrence of x cannot be below another parameter, because having x occur in one of its arguments would make that other parameter non-base-simple, contradicting the occurrence of x being leftmost. Thus x is neither base-simple nor below another parameter; so x is of functional type. Moreover, non-base-simplicity implies non-ω-simplicity. Hence, we can apply Lemma 4.21 (iteration).

2. Otherwise suppose that $\varrho_j a$ discards some of its parameters. By one of the induction invariants, $\varrho_j a$ is not an elimination variable. Hence Lemma 4.20 (elimination) applies. The newly introduced elimination variable G satisfies the required invariants, because Lemma 4.20 guarantees that $\varrho_{j+1} G$ uses its parameters and shares the body with $\varrho_j a$ which by assumption of this case contains only base-simple occurrences.

3. Otherwise every parameter of $\varrho_j a$ has occurrences and all of them are base-simple. We are going to show that Decompose is a valid transition and decreases $\varrho_j E_j$. By Lemma 4.15 we conclude from $\varrho_j u = \varrho_j v$ that $\varrho_j u_i = \varrho_j v_i$ for every i. Hence the new constraints $E_{j+1} = E_j \setminus \{u \stackrel{?}{=} v\} \cup \{u_i \stackrel{?}{=} v_i \mid \text{for all } i\}$ after Decompose are unified by ϱ_j. This allows us to define $\varrho_{j+1} = \varrho_j$ and $\sigma_{j+1} = \sigma_j$. To check that $\varrho_{j+1} E_{j+1} = \varrho_j E_{j+1}$ is smaller than $\varrho_j E_j$ it suffices to check that constraints $\varrho_j u_i \stackrel{?}{=} \varrho_j v_i$ together are smaller than $\varrho_j u \stackrel{?}{=} \varrho_j v$. Since all parameters of $\varrho_j a$ have base-simple occurrences, $\varrho_j u_i$ is a subterm of $\varrho_j u = \lambda \overline{x}.\varrho_j a(\varrho_j \overline{u})$ by Lemma 4.14. Similarly for $\varrho_j v$. It follows that $\varrho_{j+1} E_{j+1}$ is smaller than $\varrho_j E_j$. Since $\varrho_{j+1} = \varrho_j$ and $\sigma_{j+1} = \sigma_j$, the other invariants are obviously preserved.

$u \stackrel{?}{=} v$ is a flex-flex pair with different heads

5. First, suppose that $\varrho_j a$ or $\varrho_j b$ is simply projective (Definition 4.17). By the induction hypothesis, the simply projective head cannot be an identification variable. Thus Lemma 4.22 (JP-style projection) applies.

6. Otherwise suppose that $\varrho_j a$ is projective but not simply. Then the head of $\varrho_j a$ is some parameter x_k. But this occurrence cannot be ω-simple because it has arguments which cannot be bound above the head x_k. Thus Lemma 4.21 (iteration) applies. If $\varrho_j b$ is projective but not simply, the same argument applies.

7. Otherwise suppose that $\varrho_j a$, $\varrho_j b$ are in simple comparison form (Definition 4.16). By one of the induction invariants, the free variables occurring in $\varrho_j E_j$ do not occur in E_j. Thus $\varrho_j a \neq a$ and $\varrho_j b \neq b$. Then Lemma 4.24 (identification) applies.

8. Otherwise $\varrho_j a$, $\varrho_j b$ are not in simple comparison form. By Lemma 4.9 and by the definition of simple comparison form, there is some opponent pair $x_k\,\overline{r}$, b in $\varrho_j a$ and $\varrho_j b$ (after possibly swapping u and v) where either the occurrence of x_k is not ω-simple (Definition 4.11) or else x_k has another ω-simple occurrence in the body of $\varrho_j a$. Then Lemma 4.21 (iteration) applies.

$u \stackrel{?}{=} v$ is a flex-rigid pair Without loss of generality, assume that a is flex and b is rigid.

9. Suppose first that $\varrho_j a$ is projective. By one of the induction invariants, a cannot be an identification variable. Thus Lemma 4.25 (Huet-style projection) applies.
10. Otherwise $\varrho_j a$ is not projective. The head of $\varrho_j a$ must be b because b is rigid, and ϱ_j unifies u and v. As $\varrho_j a$ is not projective, that means that b is not a bound variable. Thus, b must be a constant. Then Lemma 4.23 (imitation) applies.

We have now constructed a run $(E_0, \sigma_0) \longrightarrow (E_1, \sigma_1) \longrightarrow (E_2, \sigma_2) \longrightarrow \cdots$ of the procedure. This run cannot be infinite because the measure $\mathrm{ord}(E_j, \varrho_j)$ strictly decreases as j increases. Hence, at some point we reach a j such that $E_j = \emptyset$ and $\varrho_0 X = \varrho_j \sigma_j X$ for all $X \notin V$. Therefore, $(E, \mathrm{id}) \longrightarrow^* (\emptyset, \sigma_j) \longrightarrow \sigma_j$, and $\varrho X = \tau^{-1} \varrho_j \sigma_j X$ for all $X \notin V$, completing the proof. □

4.5 A New Decidable Fragment

4

We discovered a new fragment that admits a finite CSU and a simple oracle. The oracle is based on work by Prehofer and the PT procedure [133], an adaptation of the preunification procedure by Snyder and Gallier [150] (which itself is an adaptation of Huet's procedure). PT transforms an initial multiset of constraints E_0 by applying bindings ϱ. If there is a sequence $E_0 \Longrightarrow^{\varrho_1} \cdots \Longrightarrow^{\varrho_n} E_n$ such that E_n has only flex-flex constraints, we say that PT produces a preunifier $\sigma = \varrho_n \ldots \varrho_1$ with constraints E_n. A sequence fails if $E_n = \bot$. As in the previous section, we consider all terms to be $\alpha\beta\eta$-equivalence classes with the η-long β-reduced form as their canonical representative. Unlike previously, in this section we view unification constraints $s \stackrel{?}{=} t$ as **ordered** pairs. The PT transition rules, adapted for our presentation style, are as follows:

Deletion $\quad \{s \stackrel{?}{=} s\} \uplus E \Longrightarrow^{\mathrm{id}} E$

Decomposition $\quad \{\lambda\overline{x}.\, a\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, a\,\overline{t}_m\} \uplus E \Longrightarrow^{\mathrm{id}} \{s_1 \stackrel{?}{=} t_1, \ldots, s_m \stackrel{?}{=} t_m\} \uplus E$
where a is rigid

Failure $\quad \{\lambda\overline{x}.\, a\,\overline{s} \stackrel{?}{=} \lambda\overline{x}.\, b\,\overline{t}\} \uplus E \Longrightarrow^{\mathrm{id}} \bot$
where a and b are different rigid heads

SolutionL $\quad \{\lambda\overline{x}.\, F\,\overline{x} \stackrel{?}{=} \lambda\overline{x}.\, t\} \uplus E \Longrightarrow^{\varrho} \varrho E$
where F does not occur in t, t does not have a flex head, and $\varrho = \{F \mapsto \lambda\overline{x}.\, t\}$

SolutionR $\quad \{\lambda\overline{x}.\, t \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{x}\} \uplus E \Longrightarrow^{\varrho} \varrho E$
where F does not occur in t, t does not have a flex head, and $\varrho = \{F \mapsto \lambda\overline{x}.\, t\}$

ImitationL $\quad \{\lambda\overline{x}.\, F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, \mathsf{f}\,\overline{t}_n\} \uplus E \Longrightarrow^{\varrho} \varrho(\{G_1\,\overline{s}_m \stackrel{?}{=} t_1, \ldots, G_n\,\overline{s}_m \stackrel{?}{=} t_n\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, \mathsf{f}(G_1\,\overline{x}_m)\ldots(G_n\,\overline{x}_m)\}$, $\overline{G}_n$ are correctly typed fresh variables

ImitationR $\quad \{\lambda\overline{x}.\, \mathsf{f}\,\overline{t}_n \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{s}_m\} \uplus E \Longrightarrow^{\varrho} \varrho(\{t_1 \stackrel{?}{=} G_1\,\overline{s}_m, \ldots, t_n \stackrel{?}{=} G_n\,\overline{s}_m\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, \mathsf{f}(G_1\,\overline{x}_m)\ldots(G_n\,\overline{x}_m)\}$, $\overline{G}_n$ are correctly typed fresh variables

ProjectionL $\quad \{\lambda\overline{x}.\, F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, a\,\overline{t}\} \uplus E \Longrightarrow^{\varrho} \varrho(\{s_i(G_1\,\overline{s}_m)\ldots(G_j\,\overline{s}_m) \stackrel{?}{=} a\,\overline{t}\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, x_i(G_1\,\overline{x}_m)\ldots(G_j\,\overline{x}_m)\}$, $\overline{G}_j$ are correctly typed fresh variables

ProjectionR $\quad \{\lambda\overline{x}.\, a\,\overline{t} \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{s}_m\} \uplus E \Longrightarrow^{\varrho} \varrho(\{a\,\overline{t} \stackrel{?}{=} s_i(G_1\,\overline{s}_m)\ldots(G_j\,\overline{s}_m)\} \uplus E)$
where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, x_i(G_1\,\overline{x}_m)\ldots(G_j\,\overline{x}_m)\}$, $\overline{G}_j$ are correctly typed fresh variables

The grayed constraints are required to be selected by a given selection function S. We call S *admissible* if it selects only flex-rigid constraints, and prioritizes selection of constraints applicable for Failure and Decomposition and of descendant constraints of ProjectionL and ProjectionR transitions with $j = 0$ (i.e., for x_i of base type), in that order of priority. In the remainder of this section we consider only admissible selection functions, an assumption that Prehofer also makes implicitly in his thesis. Additionally, whenever we compare multisets, we use the multiset ordering defined by Dershowitz and Manna [54]. As above, we assume that the fresh variables are taken from an infinite supply V of fresh variables that are different from the variables in the initial problem and never reused.

The following lemma states that PT is *complete for preunification*:

Lemma 4.26. *Let ϱ be a unifier of a multiset of constraints E_0. Then PT produces a preunifier σ with constraints E_n, and there exists a unifier θ of E_n such that $\varrho X = \theta\sigma X$ for all X that are not contained in the supply V of fresh variables.*

Proof. This lemma is a refinement of Lemma 4.1.7 from Prehofer's PhD thesis [133], and this proof closely follows the proof of that lemma. Compared to the lemma from Prehofer's thesis, our lemma additionally establishes the relationship of the unifier θ of the resulting flex-flex constraint set E_n with the preunifier σ and the target unifier ϱ.

We prove the lemma by induction using a well-founded measure on (ϱ, E_0). The ordering is the lexicographic comparison of the following properties:

A sum of the abstraction-free sizes of the terms ϱF for each variable F mapped by ϱ

B multiset of the sizes of constraints in E_0

Here, the *abstraction-free size* is inductively defined by afsize(F) = 1; afsize(x) = 1; afsize(f) = 1; afsize($s\,t$) = afsize(s) + afsize(t); afsize($\lambda x.\, s$) = afsize(s).

If E_0 consists only of flex-flex constraints, then we can take an empty transition sequence (i.e., σ = id) and $\theta = \varrho$. Otherwise, there exists a constraint that is selected by an admissible selection function. We show that for each such constraint, there is going to be a PT transition bringing us closer to the desired preunifier.

Let $E_0 = \{s \stackrel{?}{=} t\} \uplus E_0'$. Since $s \stackrel{?}{=} t$ is selected, at least one of s and t must have a rigid head. We distinguish several cases based on the form of $s \stackrel{?}{=} t$:

- $s \stackrel{?}{=} s$: in this case, Deletion applies. This transition does not alter ϱ, but removes an equation obtaining E_1 which is smaller than E_0 and still unifiable by ϱ. By the induction hypothesis, the preunifier is reachable.
- $\lambda\overline{x}.\, a\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, b\,\overline{t}_n$, where a and b are rigid: since ϱ is a unifier, then $a = b$, $n = m$, and Decomposition applies. Similarly to the above case, we conclude that the preunifier is reachable.
- $\lambda\overline{x}.\, F\,\overline{x} \stackrel{?}{=} \lambda\overline{x}.\, t$ where t has a rigid head and F does not occur in t: SolutionL applies. Huet showed [82, proof of L5.1] that in this case $\varrho = \varrho\sigma_1$, where $\sigma_1 = \{F \mapsto \lambda\overline{x}.\, t\}$. Let $\varrho_1 = \varrho \setminus \{F \mapsto \varrho F\}$. Then $\varrho = \varrho_1\sigma_1$. Since ϱ unifies E_0', ϱ_1 unifies $E_1 = \sigma_1 E_0'$. Clearly, ϱ_1 is smaller than ϱ. Now we can apply the induction hypothesis on ϱ_1 and E_1, from which we obtain a sequence $E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$, θ which unifies E_n and

$\sigma' = \sigma_n \dots \sigma_2$, such that $\varrho_1 X = \theta\sigma' X$ for all $X \notin V$. Finally, it is clear that the sequence $E_0 \Longrightarrow^{\sigma_1} E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$, θ, and $\sigma = \sigma'\sigma_1$ are as wanted in the lemma statement.

- $\lambda\overline{x}.\, t \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{x}$ where t has a rigid head and F does not occur in t: SolutionR applies, and the case is analogous to the previous one.

- $\lambda\overline{x}.\, F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, a\,\overline{t}$ where a is rigid: depending on the value of ϱ, we can either take an ImitationL or ProjectionL step. In either case, we show that the measure reduces. Let $\varrho F = \lambda\overline{x}_m.\, b\,\overline{u}_n$ where b is either a constant or a bound variable. If b is a constant, we choose the ImitationL step; otherwise we choose the ProjectionL step. In either case, $\sigma_1 = \{F \mapsto \lambda\overline{x}_m.\, b(G_1\,\overline{x}_m)\dots(G_n\,\overline{x}_m)\}$. Then it is easy to check that $\varrho_1 = \varrho \setminus \{F \mapsto \lambda\overline{x}_m.\, b\,\overline{u}_n\} \cup \{G_1 \mapsto \lambda\overline{x}_m.\, u_1, \dots, G_n \mapsto \lambda\overline{x}_m.\, u_n\}$ unifies E_1 created as the result of imitation or projection. Clearly, ϱ_1 has smaller abstraction-free size than ϱ. Therefore, by the induction hypothesis we obtain the transitions $E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$ and substitutions θ and $\sigma' = \sigma_n \dots \sigma_2$, where $\varrho_1 X = \theta\sigma' X$ for all $X \notin V \setminus \{G_1, \dots, G_n\}$ and θ is a unifier of E_n. Our goal is to prove $\varrho X = \theta\sigma'\sigma_1 X$ for all $X \notin V$. For variables $X \notin V$ that are not F, we have $\sigma_1 X = X$ and $\varrho X = \varrho_1 X$, which implies $\varrho X = \theta\sigma'\sigma_1 X$. For $X = F$, since ϱ_1 and $\theta\sigma'$ agree on $G_1, \dots, G_n$, we conclude that $\varrho F = \theta\sigma'\sigma_1 F$. Therefore, by taking $\sigma = \sigma'\sigma_1$, and prepending E_0 to the sequence from the induction hypothesis, we can conclude that the preunifier is reachable.

- $\lambda\overline{x}.\, a\,\overline{t} \stackrel{?}{=} \lambda\overline{x}.\, F\,\overline{s}_m$ where a is rigid: either ImitationR or ProjectionR applies and the case is analogous to the previous one. □

Prehofer showed that PT terminates for some classes of constraints. Those classes impose requirements on the free variables occurring in the constraints. In particular, he identified a class of terms we call *strictly solid*[3], which requires that all arguments of free variables are either bound variables (of arbitrary type) or ground second-order terms of base type. Another important constraint is linearity: a term is called *linear* if it contains no repeated occurrences of free variables.

Example 4.27. Let G, a, and x be of base type, and F, H, g, and y be binary. Then, the term $F\,G\,\mathsf{a}$ is not strictly solid, since F is applied to a free variable G; similarly $H\,(\lambda x.\, x)\,\mathsf{a}$ is not strictly solid as the first argument is of functional type, but it is not a bound variable; $\lambda x.\, F\,x\,\mathsf{a}$ is strictly solid, but $F\,\mathsf{a}\,(\mathsf{g}\,(\lambda y.\, y\,\mathsf{a}\,\mathsf{a})\,\mathsf{a})$ is not, as the second argument of F is a ground term of third order.

Prehofer's thesis states that PT terminates on $\{s \stackrel{?}{=} t\}$ if s is linear, s shares no free variables with t, s is strictly solid, and t is second-order. Together with completeness of PT for preunification, this result implies that the PT procedure can be used to enumerate finitely many elements of a complete set of preunifiers for this class of terms.

Prehofer focused on the preunification problem, remarking that the resulting flex-flex pairs are "intricate" [133, Sect. 5.2.2]. We discovered that these flex-flex pairs actually have an MGU, allowing us to solve the full unification problem, rather than the preunification problem. Moreover, we lift the order restriction imposed by Prehofer: We identify a class

[3]In Prehofer's thesis this class is described in the statement of Theorem 5.2.6.

of terms called *solid* which requires that all arguments of free variables are either bound variables (of arbitrary type) or ground (arbitrary-order) terms of base type.

Example 4.28. Consider the setting of Example 4.27. The terms $F\,G\,\mathsf{a}$ and $H\,(\lambda x.\,x)\,\mathsf{a}$ are not solid, for the same reasons they are not strictly solid. However, since the order restriction is lifted, $F\,\mathsf{a}\,(\mathsf{g}\,(\lambda y.\,y\,\mathsf{a}\,\mathsf{a})\,\mathsf{a})$ is solid.

Put differently, we extend the preunification decidability result for linear strictly solid terms along two axes: We create an oracle for the full unification problem and we lift the order restrictions. To enumerate a CSU for a problem $E_0 = \{s \stackrel{?}{=} t\}$, where s and t are solid, s is linear and shares no free variables with t, our oracle applies the following two steps:

1. Apply the PT procedure on E_0 to obtain a preunifier σ with flex-flex constraints E_n.
2. If E_n is empty, return σ. Otherwise, choose a flex-flex constraint $u \stackrel{?}{=} v$ from E_n, and let the MGU of $u \stackrel{?}{=} v$ be ϱ. Then, set $E_n := \varrho(E_n \setminus \{u \stackrel{?}{=} v\})$ and $\sigma := \varrho\sigma$, and repeat step 2.

To show that this oracle terminates and yields a CSU, we must prove that the PT procedure terminates on the above described class of problems (Lemma 4.31). Moreover, we must show how to compute MGUs for the remaining flex-flex pairs (Lemma 4.32 and 4.33). Our results are combined together in Theorem 4.34. Note that we use names Solution, Imitation, and Projection when only one orientation of the corresponding rule is applicable.

Toward proving that the PT procedure terminates on the above described class of problems, we first consider the corresponding matching problem. A *matching problem* is a unification problem $\{s \stackrel{?}{=} t\}$ where t is ground. In what follows, we establish some useful properties of matching problems in which both s and t are solid (*solid matching problems*). We call the unifier of a matching problem a *matcher*.

Lemma 4.29. *PT terminates on every solid matching problem* $\{s \stackrel{?}{=} t\}$.

Proof. We show termination by designing a measure on a matching problem E that decreases with each application of a PT transition (possibly followed by Decomposition or Failure steps). Our measure function compares the following properties lexicographically:

A multiset of the sizes of right-hand sides of the constraints in E

B number of free variables in E

Clearly, when applying PT transitions, the problem stays a matching problem because the applied bindings do not introduce free variables on the right-hand sides. It is also easy to check that each applied binding keeps the terms in the solid fragment. Namely, bindings for both Imitation and Projection transitions are patterns, which means that, after applying a binding, fresh free variables are applied only to bound variables or ground base-type terms. The Solution transition effectively replaces a variable with a ground term, which is obviously solid. We show each transition either trivially terminates or reduces the measure:

Deletion A decreases.

Decomposition A decreases.

Failure Trivial—represents a terminal node.

Solution This transition applies on constraints of the form $F \stackrel{?}{=} t$. This reduces A, since the constraint $F \stackrel{?}{=} t$ is removed and F cannot appear on the right-hand side.

Imitation The rule is applicable only on $\lambda\overline{x}.F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\mathsf{f}\,\overline{t}_n$. The Imitation transition replaces the constraint with $\{H_i\,\overline{s}_m \stackrel{?}{=} \overline{t}_i \mid 1 \le i \le n\}$. This reduces A, as F does not appear on any right-hand side.

Projection The rule is applicable only on $\lambda\overline{x}.F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.a\,\overline{t}_n$. If a is a bound variable, and we project F to argument s_i different from a, Failure applies and PT trivially terminates. If we project F to a, we apply Decomposition, which reduces A. If a is a constant, for Failure not to apply, we have to project to a base-type ground term s_i. This does not increase A, since no variables appear on right-hand sides, but removes the variable F from E, reducing B by one. □

Recall that we say σ is a grounding substitution if for every variable F mapped by σ, σF is ground (Sect. 2.2).

Lemma 4.30. *All unifiers produced by PT for the solid matching problem $\{s \stackrel{?}{=} t\}$ are grounding substitutions.*

Proof. Closely following the proof of Lemma 5.2.5 in Prehofer's PhD thesis [133], we prove our claim by induction on the length of the PT transition sequence that leads to the unifier. We know this sequence is finite by Lemma 4.29. The base case of induction, for the empty sequence, is trivial. The induction step is made using one of the following transitions:

Deletion Trivial.

Decomposition Trivial.

Failure This rule is not relevant, since it will not lead to the unifier.

Solution This rule applies the substitution $\{F \mapsto t\}$, which is grounding since t is ground.

Imitation This transition applies on constraints of the form

$$\lambda\overline{x}.F\,\overline{s}_k \stackrel{?}{=} \lambda\overline{x}.\mathsf{f}\,\overline{t}_l$$

The binding for Imitation is $\varrho = \{F \mapsto \lambda\overline{x}_k.\mathsf{f}(G_1\,\overline{s}_k)\ldots(G_l\,\overline{s}_k)\}$, and reduces the problem to $\{G_i\,\overline{s}_k \stackrel{?}{=} t_i \mid 1 \le i \le l\}$. Since the right-hand sides are ground, any unifier σ produced by PT must map all of the variables $G_1,\ldots,G_l$. By the induction hypothesis, σG_i is ground. Therefore, $\sigma\varrho$ must map F to a ground term.

Projection This transition applies on constraints of the form

$$\lambda\overline{x}.F\,\overline{s}_k \stackrel{?}{=} \lambda\overline{x}.t$$

The binding for Projection is $\varrho = \{F \mapsto \lambda\overline{x}_k.x_i(G_1\,\overline{x}_k)\ldots(G_j\,\overline{x}_k)\}$, and reduces the problem to $\{s_i(G_1\,\overline{s}_k)\ldots(G_j\,\overline{s}_k) \stackrel{?}{=} a\,\overline{t}_l\}$. Since the right-hand side is ground, any unifier σ produced by PT must map all of the variables $G_1,\ldots,G_j$. By the induction hypothesis, σG_i is ground. Therefore, $\sigma\varrho$ must map F to a ground term. □

Lemma 4.31. *If s and t are solid, and s is linear and shares no free variables with t, then PT terminates for the preunification problem $\{s \stackrel{?}{=} t\}$, and all remaining flex-flex constraints are solid.*

Proof. This lemma is a modification of Lemma 5.2.1 and Theorem 5.2.6 from Prehofer's PhD thesis [133]. Correspondingly, the following proof closely follows the proofs for these two lemmas. We also adopt the notion of an *isolated* variable from Prehofer's thesis: a variable is isolated in a multiset E of constraints if it appears exactly once in E.

First, similar to the proof of previous lemma we conclude each transition maintains the condition that the terms remain solid. Second, we have to show that variables on left-hand sides remain isolated. For ImitationL(R) and ProjectionL(R) rules, the preservation of this invariant is obvious. Later, we also show that SolutionL(R) preserves it. Third, since s and t share no variables, and s is linear, no rule can introduce a variable from the right-hand side to the left-hand side.

To prove termination of PT, we devise a measure that decreases with each application of a PT transition. The measure lexicographically compares the following properties:

A number of occurrences of constant symbols and bound variables on left-hand sides that are not below free variables

B number of free variables on right-hand sides

C multiset of the sizes of right-hand sides

We show that each transition either trivially terminates or reduces the measure:

Deletion A does not increase and at least one of A or B reduces.

Decomposition A reduces.

Failure Trivial.

SolutionL $F \stackrel{?}{=} t$: since F is isolated, A is unchanged, B is not increased, and C reduces. All variables on left-hand sides remain isolated since they are unaffected by this substitution.

SolutionR $t \stackrel{?}{=} F$: since F does not occur on any left-hand side and the head of t must be a constant or bound variable (otherwise the rule would not be applicable), A reduces. Even though t might contain some free variables and F can have multiple occurrences on right-hand sides, all free variables on left-hand sides remain isolated. Namely, all free variables in t occur exactly once in the multiset, and since $t \stackrel{?}{=} F$ is removed, all of them either disappear or end up on right-hand sides. We distinguish the following two forms of the selected constraint:

ImitationL $\lambda\overline{x}.F\,\overline{v}_n \stackrel{?}{=} \lambda\overline{x}.\mathsf{f}\,\overline{u}_m$: We replace this constraint by $\{H_i\,\overline{v}_n \stackrel{?}{=} u_i \mid 1 \le i \le m\}$ and apply the ImitationL binding $F \mapsto \lambda\overline{x}_n.\mathsf{f}(H_1\,\overline{x}_n)\ldots(H_m\,\overline{x}_n)$. As F is isolated, A and B do not increase. Since F is isolated, it does not occur on any right-hand side, and hence C decreases.

ImitationR $\lambda\overline{x}.\,f\,\overline{u}_m \stackrel{?}{=} \lambda\overline{x}.\,F\,\overline{v}_n$: applying the ImitationR transition as above will reduce A since F cannot appear on any left-hand side.

ProjectionL $\lambda\overline{x}.\,F\,\overline{v}_n \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{u}_m$: if a is a bound variable, then for Failure not to be applicable afterward, we have to project F onto argument v_i equal to a. Then we proceed like for ImitationL. If a is not a bound variable, then we have to project to some base-type term v_j (otherwise Failure would be applicable afterward). This reduces the problem to $\lambda\overline{x}.\,v_j \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{u}_m$. This is a solid matching problem, whose solutions computed by PT are grounding substitutions (see Lemma 4.30). Applying one of those solutions will eliminate all the free variables in $\lambda\overline{x}.\,a\,\overline{u}_m$. Since PT is parameterized by an admissible selection function, we know that there are no constraints descending from a simple projection in E since the constraint $\lambda\overline{x}.\,F\,\overline{v}_n \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{u}_m$ was chosen, which, due to solidity restrictions, cannot be such a descendant. Therefore, we know that PT will transform the descendants of the matching problem $\lambda\overline{x}.\,v_j \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{u}_m$ until either Failure is observed (making PT trivially terminating) or no descendant exists and the grounding matcher is computed (see Lemmas 4.30 and 4.29). This results in removal of the original constraint $\lambda\overline{x}.\,F\,\overline{v}_n \stackrel{?}{=} \lambda\overline{x}.\,a\,\overline{u}_m$ and application of the computed grounding matcher, which will either remove all the free variables in the right-hand side of the constraint, not increasing A and reducing B, or, if no free variables occur in the right-hand side, reduce C while not increasing A and B.

ProjectionR $\lambda\overline{x}.\,a\,\overline{v}_n \stackrel{?}{=} \lambda\overline{x}.\,F\,\overline{u}_m$: if a is a bound variable, projecting F onto argument u_i will either enable application of Decomposition as the next step, reducing A, or it will result in Failure, trivially terminating. If a is a constant, then projecting F onto some u_j will either yield Failure or enable Decomposition, reducing A. □

Enumerating a CSU for a solid flex-flex pair may seem as hard as for any other flex-flex pair; however, the following two lemmas show that solid pairs admit an MGU:

Lemma 4.32. *The unification problem* $\{\lambda\overline{z}.\,F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{z}.\,F\,\overline{s'}_m\}$*, where both terms are solid, has an MGU of the form* $\sigma = \{F \mapsto \lambda\overline{x}_m.\,G\,x_{j_1}\ldots x_{j_r}\}$ *where* G *is an auxiliary variable, and* $1 \le j_1 < \cdots < j_r \le m$ *are exactly those indices* j_i *for which* $s_{j_i} = s'_{j_i}$*.*

Proof. Let ϱ be a unifier for the given unification problem. Let $\lambda\overline{x}.\,u = \varrho F$. Take an arbitrary subterm of u whose head is a bound variable x_i. If x_i is of function type, it corresponds to either s_i or s'_i which, due to solidity restrictions, has to be a bound variable. Furthermore, since ϱ is a unifier, s_i and s'_i have to be the syntactically equal. Similarly, if x_i is of base type, it corresponds to two ground terms s_i and s'_i which have to be syntactically equal. We conclude that ϱ can use variables from $\overline{x}_n$ only if they correspond to syntactically equal terms. Therefore, there is a substitution θ such that $\varrho X = \theta\sigma X$ for all $X \neq G$. Due to the arbitrary choice of ϱ, we conclude that σ is an MGU. □

Lemma 4.33. *Let* $\{\lambda\overline{x}.\,F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\,F'\,\overline{s'}_{m'}\}$ *be a solid unification problem where* $F \neq F'$*. By Lemma 4.29, there exists a finite CSU* $\{\sigma_i^1, \ldots, \sigma_i^{k_i}\}$ *of the problem* $\{s_i \stackrel{?}{=} H_i\,\overline{s'}_{m'}\}$*, where* H_i *is a fresh free variable. Let* $\lambda\overline{y}_{m'}.\,s_i^j = \lambda\overline{y}_{m'}.\,\sigma_i^j(H_i)\,\overline{y}_{m'}$*. Similarly, also by Lemma 4.29, there exists a finite CSU* $\{\tilde{\sigma}_i^1, \ldots, \tilde{\sigma}_i^{l_i}\}$ *of the problem* $\{s'_i \stackrel{?}{=} \tilde{H}_i\,\overline{s}_m\}$*, where* $\tilde{H}_i$ *is a fresh free variable. Let* $\lambda\overline{x}_m.\,{s'}_i^j = \lambda\overline{x}_m.\,\tilde{\sigma}_i^j(\tilde{H}_i)\,\overline{x}_m$*. Let* Z *be a fresh free variable. An MGU* σ *for the given problem is*

$$F \mapsto \lambda\overline{x}_m.\, Z\, \underbrace{x_1 \dots x_1}_{k_1 \text{ times}} \dots \underbrace{x_m \dots x_m}_{k_m \text{ times}}\, {s'}_1^1 \dots {s'}_1^{l_1} \dots {s'}_{m'}^1 \dots {s'}_{m'}^{l_{m'}}$$

$$F' \mapsto \lambda\overline{y}_{m'}.\, Z\, s_1^1 \dots s_1^{k_1} \dots s_m^1 \dots s_m^{k_m}\, \underbrace{y_1 \dots y_1}_{l_1 \text{ times}} \dots \underbrace{y_{m'} \dots y_{m'}}_{l_{m'} \text{ times}}$$

where the auxiliary variables are $H_1, \dots, H_m$, $\tilde{H}_1, \dots \tilde{H}_{m'}$, Z and all auxiliary variables associated with the above CSUs.

Proof. Let ϱ be an arbitrary unifier for the problem $\lambda\overline{x}.\, F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.\, F'\,\overline{s}'_{m'}$. We prove σ is an MGU by showing that there exists a substitution θ such that $\varrho X = \theta\sigma X$ for all nonauxiliary variables X. We can focus only on $X \in \{F, F'\}$ because all other nonauxiliary variables appear neither in the original problem nor in σF or $\sigma F'$; so we can simply define $\theta X = \varrho X$.

Let $\lambda\overline{x}_m.\, u = \varrho F$ and $\lambda\overline{y}_{m'}.\, u' = \varrho F'$, where the bound variables $\overline{x}_m$ and $\overline{y}_{m'}$ have been α-renamed apart. We also assume that the names of variables bound inside u and u' are α-renamed so that they are different from $\overline{x}_m$ and $\overline{y}_{m'}$. Finally, bound variables from the definition of σ have been α-renamed to match $\overline{x}_m$ and $\overline{y}_{m'}$. We define θ to be the substitution

$$\theta = \{Z \mapsto \lambda z_1^1 \dots z_1^{k_1} \dots z_m^1 \dots z_m^{k_m}\, w_1^1 \dots w_1^{l_1} \dots w_{m'}^1 \dots w_{m'}^{l_{m'}}.\, \mathsf{diff}(u, u')\}$$

where $\mathsf{diff}(v, v')$ is defined recursively as

$$\mathsf{diff}(\lambda x.\, v, \lambda y.\, v') = \lambda x.\, \mathsf{diff}(v, \{y \mapsto x\}v') \tag{4.1}$$

$$\mathsf{diff}(a\,\overline{v}_n, a\,\overline{v'}_n) = a\,\mathsf{diff}(v_1, v'_1) \dots \mathsf{diff}(v_n, v'_n) \tag{4.2}$$

$$\mathsf{diff}(x_i, v') = z_i^k, \text{ if } v' = s_i^k \tag{4.3}$$

$$\mathsf{diff}(v, y_i) = w_i^l, \text{ if } v = {s'}_i^l \tag{4.4}$$

$$\mathsf{diff}(x_i\,\overline{v}_n, y_j\,\overline{v'}_n) = z_i^k\,\mathsf{diff}(v_1, v'_1) \dots \mathsf{diff}(v_n, v'_n), \text{ if } y_j = s_i^k \tag{4.5}$$

From diff's definition it is clear that there are terms v, v' for which it is undefined. However, we will show that for each u and u' that are bodies of bindings from a unifier ϱ, diff is defined and has the desired property. In equations (4.3), (4.4), and (4.5), if there are multiple numbers k or l that fulfill the condition, choose an arbitrary one. We need to show that $\varrho F = \theta\sigma F$ and $\varrho F' = \theta\sigma F'$. By the definitions of u, u', θ and σ and β-reduction, this is equivalent to

$$\lambda\overline{x}_m.\, u = \lambda\overline{x}_m.\{z_i^k \mapsto x_i, w_i^l \mapsto {s'}_i^l \text{ for all } k, l, i\}\mathsf{diff}(u, u')$$
$$\lambda\overline{y}_{m'}.\, u' = \lambda\overline{y}_{m'}.\{z_i^k \mapsto s_i^k, w_i^l \mapsto y_i \text{ for all } k, l, i\}\mathsf{diff}(u, u')$$

We will show by induction that for any $\lambda\overline{x}_m.\, v$, $\lambda\overline{y}_m.\, v'$ such that

$$\{x_1 \mapsto s_1, \dots, x_m \mapsto s_m\}v = \{y_1 \mapsto s'_1, \dots, y_{m'} \mapsto s'_{m'}\}v' \tag{$\star$}$$

we have

$$\begin{aligned} v &= \{z_i^k \mapsto x_i, w_i^l \mapsto {s'}_i^l \text{ for all } k, l, i\}\mathsf{diff}(v, v') \\ v' &= \{z_i^k \mapsto s_i^k, w_i^l \mapsto y_i \text{ for all } k, l, i\}\mathsf{diff}(v, v') \end{aligned} \tag{$\dagger$}$$

The equation ($\star$) holds for $v = u$ and $v' = u'$ because ϱ is a unifier of $\lambda\overline{x}.F\,\overline{s}_m \stackrel{?}{=} \lambda\overline{x}.F'\,\overline{s'}_{m'}$. Therefore, once we have shown that ($\star$) implies (†), we know that (†) holds for $v = u$ and $v' = u'$ and we are done.

We prove that ($\star$) implies (†) by induction on the sizes of v and v'. We consider the following cases:

$v = \lambda x.v_1$ For ($\star$) to hold, v and v' must be of the same type. Therefore, the λ-prefixes of their η-long representatives must have the same length and we can apply equation 4.1. By the induction hypothesis, (†) holds.

$v = x_i$ In this case, $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}v = s_i$. Since ($\star$) holds, v' must be an instance of a unifier from the CSU of $s_i = H_i\,\overline{s'}_{m'}$. However, since s_i and all terms in $\overline{s'}_{m'}$ are ground, $\lambda\overline{y}_m.v' = \sigma_i^k(H_i)$, for some k. Then, $\mathsf{diff}(x_i, v') = z_i^k$, and it is easy to check that (†) holds.

$v = x_i\,\overline{v}_n, n > 0$ In this case, x_i is mapped to s_i which, due to solidity restrictions, has to be a functional bound variable. Since ($\star$) holds, we conclude that the head of $\{y_1 \mapsto s'_1, \ldots, y_{m'} \mapsto s'_{m'}\}v'$ must be s'_j, such that $s'_j = s_i$; this also means that $v' = y_j\,\overline{v'}_n$. Therefore, it is easy to check that some $\tau = \{H_i \mapsto \lambda\overline{y}_{m'}.y_j\}$ is a matcher for the problem $s_i = H_i\,\overline{s'}_{m'}$. For some k, $\sigma_i^k = \tau$, i.e., $\mathsf{diff}(v, v') = z_i^k\,\mathsf{diff}(v_1, v'_1)\ldots\mathsf{diff}(v_n, v'_n)$. By the induction hypothesis, we get that (†) holds.

$v = a\,\overline{v}_n$ In the remaining cases a is either a free variable, a loose bound variable different from $x_1, \ldots, x_m$, or a constant. If a is a free variable or a loose bound variable different from $x_1, \ldots, x_m$, then $v' = a\,\overline{v'}_n$, since ($\star$) holds, all of $\overline{s'}_{m'}$ are ground, and bound variables different from $x_1, \ldots, x_m$ and $y_1, \ldots, y_{m'}$ are renamed to match by equation 4.1. By the induction hypothesis and by equation 4.2, we obtain (†). If a is a constant, we consider two cases: either $v' = a\,\overline{v'}_n$, which allows us to apply the induction hypothesis and obtain (†) as above, or $v' = y_j\,\overline{v'}_m$. Since a is a constant, y_j cannot be a functional bound variable, since then it would be mapped to s'_j, which due to solidity restrictions also has to be a functional bound variable and ($\star$) would not hold. Therefore $v' = y_j$. In this case, we proceed as in the case $v = x_i$ with the roles of v and v' swapped. □

Theorem 4.34. *Let s and t be solid terms that share no free variables, and let s be linear. Then the unification problem $\{s \stackrel{?}{=} t\}$ has a finite CSU.*

Proof. By Lemma 4.31, PT terminates on $\{s \stackrel{?}{=} t\}$ with a finite set of preunifiers σ, each associated with a multiset E of solid flex-flex pairs.

An MGU δ_E of the remaining multiset E of solid flex-flex constraints can be found as follows. Choose one constraint $(u \stackrel{?}{=} v) \in E$ and determine an MGU ϱ for it using Lemma 4.32 or 4.33. Then the set $\varrho(E \setminus \{u \stackrel{?}{=} v\})$ also contains only solid flex-flex constraints, and we iterate this process by choosing a constraint from $\varrho(E \setminus \{u \stackrel{?}{=} v\})$ next until there are no constraints left, eventually yielding an MGU ϱ' of $\varrho(E \setminus \{u \stackrel{?}{=} v\})$. Finally, we obtain the MGU $\delta_E = \varrho'\varrho$ of E.

Let $U = \{\delta_E\sigma \mid \text{PT produces preunifier } \sigma \text{ with constraints } E\}$. By termination of PT, U is finite. We show that U is a CSU. Let ϱ be an arbitrary unifier for $\{s \stackrel{?}{=} t\}$. By Lemma 4.26,

PT produces a preunifier σ with flex-flex constraints E such that there is a unifier θ of E and $\varrho X = \theta\sigma X$ for all X not contained in the supply of fresh variables V. Since δ_E is an MGU of E, assuming that we use variables from V in the role of the auxiliary variables, there exists a substitution θ' such that $\theta X = \theta'\delta_E X$ for all $X \notin V$. If we make sure that we never reuse fresh variables and that the supply V does not contain any variables from the initial problem, it follows that $\varrho X = \theta'\delta_E\sigma X$ for all $X \notin V$. Therefore, U is a CSU. □

The proof of Theorem 4.34 provides an effective way to calculate a CSU using PT and the results of Lemmas 4.32 and 4.33.

Example 4.35. Let $\{F(\mathsf{fa}) \stackrel{?}{=} \mathsf{g}\,\mathsf{a}(G\mathsf{a})\}$ be the unification problem to solve. Projecting F onto the first argument will lead to a nonunifiable problem, so the imitation of g is performed, yielding a binding $\sigma_1 = \{F \mapsto \lambda x.\mathsf{g}(F_1\,x)(F_2\,x)\}$. After decomposition, this yields the problem $\{F_1(\mathsf{fa}) \stackrel{?}{=} \mathsf{a}, F_2(\mathsf{fa}) \stackrel{?}{=} G\mathsf{a}\}$. Again, a can only be imitated for F_1—building a new binding $\sigma_2 = \{F_1 \mapsto \lambda x.\mathsf{a}\}$. Finally, this yields the problem $\{F_2(\mathsf{fa}) \stackrel{?}{=} G\mathsf{a}\}$. According to Lemma 4.33, CSUs for the problems $J_1\,\mathsf{a} = \mathsf{fa}$ and $I_1(\mathsf{fa}) \stackrel{?}{=} \mathsf{a}$ are found using PT. The latter problem has a singleton CSU $\{I_1 \mapsto \lambda x.\mathsf{a}\}$, whereas the former has a CSU containing $\{J_1 \mapsto \lambda x.\mathsf{f}\,x\}$ and $\{J_1 \mapsto \lambda x.\mathsf{fa}\}$. Combining these solutions, an MGU $\sigma_3 = \{F_2 \mapsto \lambda x.H\,x\,x\,\mathsf{a}, G \mapsto \lambda x.H(\mathsf{fa})(\mathsf{f}\,x)\,x\}$ is obtained for $F_2(\mathsf{fa}) \stackrel{?}{=} G\mathsf{a}$. Finally, the MGU of the original problem is computed: $\sigma = \sigma_3\sigma_2\sigma_1 = \{F \mapsto \lambda x.\mathsf{g}\,\mathsf{a}(H\,x\,x\,\mathsf{a}), G \mapsto \lambda x.H(\mathsf{fa})(\mathsf{f}\,x)\,x\}$. For brevity, we omitted the intermediate bindings of auxiliary variables in σ.

The solid fragment is useful for automatic theorem provers based on λ-superposition [18]. As Example 4.35 shows, when the solid oracle is used, superposing from $F(\mathsf{fa})$ into $\mathsf{g}\,\mathsf{a}(G\mathsf{a})$ yields a single clause; without it, our procedure does not terminate.

Small examples that violate conditions of Theorem 4.34 and admit only infinite CSUs can easily be found. The problem $\{\lambda x.F(\mathsf{f}\,x) \stackrel{?}{=} \lambda x.\mathsf{f}(F\,x)\}$ violates variable distinctness and is a well-known example of a problem with only infinite CSUs. Similarly, $\lambda x.\mathsf{g}(F(\mathsf{f}\,x))\,F \stackrel{?}{=} \lambda x.\mathsf{g}(\mathsf{f}(G\,x))\,G$, which violates linearity, reduces to the previous problem. Only ground arguments to free variables are allowed because $\{F\,X \stackrel{?}{=} G\mathsf{a}\}$ has only infinite CSUs. Finally, it is crucial that functional arguments to free variables are only bound variables: The problem $\{\lambda y.X(\lambda x.x)\,y \stackrel{?}{=} \lambda y.y\}$ has only infinite CSUs.

4.6 An Extension of Fingerprint Indexing

A fundamental building block for almost all automated reasoning tools is the operation of retrieving term pairs that satisfy certain conditions, e.g., unifiable terms, instances or generalizations. Indexing data structures are used to implement this operation efficiently. If the data structure retrieves precisely the terms that satisfy the condition, it is called *perfect*.

Higher-order indexing has received little attention compared to its first-order counterpart. However, recent research in higher-order theorem proving increased the interest in higher-order indexing [24, 106]. Recall from Chapter 3 that a *fingerprint index* [144] is an imperfect index based on the idea that the skeleton of the term consisting of all nonvariable positions is not affected by substitutions. Therefore, we can easily determine that the terms are not unifiable (or matchable) if they disagree on a fixed set of sample positions.

More formally, when we sample an untyped first-order term t on a sample position p, the *generic fingerprinting function* gfpf distinguishes four possibilities:

$$\mathrm{gfpf}(t,p) = \begin{cases} \mathsf{f} & \text{if } t|_p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|_p \text{ is a variable} \\ \mathsf{B} & \text{if } t|_q \text{ is a variable for some proper prefix } q \text{ of } p \\ \mathsf{N} & \text{otherwise} \end{cases}$$

We define the *fingerprinting function* $\mathrm{fp}(t) = (\mathrm{gfpf}(t,p_1),\ldots,\mathrm{gfpf}(t,p_n))$, based on a fixed tuple of positions $\overline{p}_n$. Determining whether two terms are compatible for a given retrieval operation reduces to checking their fingerprints' componentwise compatibility. The following matrices determine the compatibility for retrieval operations:

	f_1	f_2	A	B	N
f_1		✗			✗
f_2	✗				✗
A					✗
B					
N	✗	✗	✗		

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
f_2	✗		✗	✗	✗
A				✗	✗
B					
N	✗	✗	✗	✗	

The left matrix determines unification compatibility, while the right matrix determines compatibility for matching term s (rows) onto term t (columns). Symbols f_1 and f_2 stand for arbitrary distinct constants. Incompatible features are marked with ✗. For example, given a tuple of term positions $(1, 1.1.1, 2)$, and terms $\mathsf{f}(\mathsf{g}(X),\mathsf{b})$ and $\mathsf{f}(\mathsf{f}(\mathsf{a},\mathsf{a}),\mathsf{b})$, their fingerprints are $(\mathsf{g},\mathsf{B},\mathsf{b})$ and $(\mathsf{f},\mathsf{N},\mathsf{b})$, respectively. Since the first fingerprint component is incompatible, terms are not unifiable.

Fingerprints for the terms in the index are stored in a trie data structure. This allows us to efficiently filter out terms that are not compatible with a given retrieval condition. For the remaining terms, a unification or matching procedure must be invoked to determine whether they satisfy the condition or not.

The fundamental idea of first-order fingerprint indexing carries over to higher-order terms—application of a substitution does not change the rigid skeleton of a term. However, to extend fingerprint indexing to higher-order terms, we must address the issues of $\alpha\beta\eta$-normalization and figure out how to cope with λ-abstractions and bound variables. To that end, we define a function $\lfloor t \rfloor$, defined on η-long β-reduced terms in De Bruijn [42] notation:

$$\lfloor F\,\overline{s} \rfloor = F \qquad \lfloor \mathbf{i}\,\overline{s}_n \rfloor = \mathsf{db}_i^\alpha(\lfloor s_1 \rfloor,\ldots,\lfloor s_n \rfloor) \qquad \lfloor \mathsf{f}\,\overline{s}_n \rfloor = \mathsf{f}(\lfloor s_1 \rfloor,\ldots,\lfloor s_n \rfloor) \qquad \lfloor \lambda\overline{x}.\, s \rfloor = \lfloor s \rfloor$$

We let $\mathbf{i}$ be a bound variable of type α with De Bruijn index i, and db_i^α be a fresh constant corresponding to this variable. All constants db_i^α must be fresh. Effectively, $\lfloor\ \rfloor$ transforms an η-long β-reduced higher-order term to an untyped first-order term. Let $t\downarrow_{\beta\eta}$ be the η-long β-reduced form of t; the higher-order generic fingerprinting function $\mathrm{gfpf}_{\mathsf{ho}}$, which relies on conversion $\langle t \rangle_{\mathsf{db}}$ from named to De Bruijn representation, is defined as

$$\mathrm{gfpf}_{\mathsf{ho}}(t,p) = \mathrm{gfpf}(\lfloor \langle t\downarrow_{\beta\eta}\rangle_{\mathsf{db}} \rfloor, p)$$

If we define $\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor\langle t \downarrow_{\beta\eta}\rangle_{\mathsf{db}}\rfloor)$, we can support fingerprint indexing for higher-order terms with no changes to the compatibility matrices. For example, consider the terms $s = (\lambda x\, y.\, x\, y)\,\mathsf{g}$ and $t = \mathsf{f}$, where g has the type $\alpha \to \beta$ and f has the type $\alpha \to \alpha \to \beta$. For the tuple of positions $(1, 1.1.1, 2)$ we get

$$\mathrm{fp}_{\mathsf{ho}}(s) = \mathrm{fp}(\lfloor\langle s \downarrow_{\beta\eta}\rangle_{\mathsf{db}}\rfloor) = \mathrm{fp}(\mathsf{g}(\mathsf{db}_0^\alpha)) = (\mathsf{db}_0^\alpha, \mathsf{N}, \mathsf{N})$$
$$\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor\langle t \downarrow_{\beta\eta}\rangle_{\mathsf{db}}\rfloor) = \mathrm{fp}(\mathsf{f}(\mathsf{db}_1^\alpha, \mathsf{db}_0^\alpha)) = (\mathsf{db}_1^\alpha, \mathsf{N}, \mathsf{db}_0^\alpha)$$

As the first and third fingerprint component are incompatible, the terms are not unifiable.

Other first-order indexing techniques such as feature vector indexing and substitution trees can probably be extended to higher-order terms using the method described here.

4.7 Implementation

As described in Sect. 2.5.7, Zipperposition [48, 49] is an open-source[4] theorem prover written in OCaml. It is a versatile testbed for prototyping extensions to superposition-based theorem provers. It was initially designed as a prover for rank-1 polymorphic [31] first-order logic and then extended to higher-order logic. A recent addition is a complete mode for Boolean-free higher-order logic [18], which depends on a unification procedure that can enumerate a CSU. We implemented our procedure in Zipperposition.

We used OCaml's functors to create a modular implementation. The core of our procedure is implemented in a module that is parameterized by another module providing oracles and implementing the Bind step. In this way we obtain the complete or pragmatic procedure and seamlessly integrate oracles while reusing as much common code as possible.

To enumerate all elements of a possibly infinite CSU, we rely on lazy lists whose elements are subsingletons of unifiers (either one-element sets containing a unifier or empty sets). The search space must be explored in a *fair* manner, meaning that no branch of the constructed tree is indefinitely postponed.

Each Bind step will give rise to new unification problems $E_1, E_2, \ldots$ to be solved. Solutions to each of those problems are lazy lists $p_1, p_2, \ldots$ containing subsingletons of unifiers. To avoid postponing some unifier indefinitely, we use the dovetailing technique: we first take one subsingleton from p_1, then one from each of p_1 and p_2. We continue with one subsingleton from each of p_1, p_2, and p_3, and so on. Empty lazy lists are ignored in the traversal. To ensure we do not remain stuck waiting for a unifier from a particular lazy list, the procedure periodically returns an empty set, indicating that the next lazy list should be probed.

The implemented selection function for our procedure prioritizes selection of rigid-rigid over flex-rigid pairs, and flex-rigid over flex-flex pairs. However, since the constructed substitution σ is not applied eagerly, heads can appear to be flex, even if they become rigid after dereferencing and normalization. To mitigate this issue, we dereference the heads with σ, but do not normalize, and use the resulting heads for prioritization.

We implemented oracles for the pattern, solid, and fixpoint fragment. Fixpoint unification [82] is concerned with problems of the form $\{F \stackrel{?}{=} t\}$. If F does not occur in t, $\{F \mapsto t\}$ is an MGU for the problem. If there is a position p in t such that $t|_p = F\,\overline{u}_m$ and for each

[4] https://github.com/sneeuwballen/zipperposition

prefix $q \neq p$ of p, $t|_q$ has a rigid head and either $m = 0$ or t is not a λ-abstraction, then we can conclude that $F \stackrel{?}{=} t$ has no solutions. Otherwise, the fixpoint oracle is not applicable.

For second-order logic with only unary constants, it is decidable whether a unifier for a problem in this class (called *monadic second-order*) exists [61]. As this class of terms admits a possibly infinite CSU, this oracle cannot be used for OracleSucc, but it can be used for OracleFail. Similarly the fragment of second-order terms with no repeated occurrences of free variables has decidable unifier existence but possibly infinite CSUs [57]. Due to their limited applicability and high complexity we decided not to implement these oracles.

4.8 Evaluation

We evaluated the implementation of our unification procedure in Zipperposition, assessing the complete variant and the pragmatic variant, the latter with several different combinations of limits for number of bindings. As part of the implementation of the complete mode for Boolean-free higher-order logic in Zipperposition [18], Bentkamp implemented a straightforward version of the JP procedure. This version is faithful to the original description, with a check as to whether a (sub)problem can be solved using a first-order oracle as the only optimization. Our evaluations were performed on StarExec Miami [154] servers with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz with 60 s CPU limit.

Contrary to first-order unification, there is no widely available benchmark set designed solely for evaluating performance of higher-order unification algorithms. Thus, we used all 2606 monomorphic higher-order theorems from the TPTP 7.2.0 library [157] and 832 monomorphic higher-order Sledgehammer (SH) generated problems [156] as our benchmarks.[5] Many TPTP problems require synthesis of complicated unifiers, whereas SH problems are only mildly higher-order—many of them are solved with first-order unifiers.

We used the naive implementation of the JP procedure (**jp**) as a baseline to evaluate the performance of our procedure. We compare it with the complete variant of our procedure (**cv**) and pragmatic variants (**pv**) with several different configurations of limits for applied bindings. All other Zipperposition parameters have been fixed to the values of a variant of a well-performing configuration we used for the 2019 installment of THF division of the CASC theorem proving competition [160]. The cv configuration and all of the pv configurations use only pattern unification as an underlying oracle. To test the effect of oracle choice, we evaluated the complete variant in eight combinations: with no oracles (**n**), with only fixpoint (**f**), pattern (**p**), or solid (**s**) oracle, and with their combinations: **fp**, **fs**, **ps**, **fps**.

Figure 4.1 compares different variants of the procedure with the naive JP implementation. Each pv configuration is denoted by pv^a_{bcde} where a is the limit on the total number of applied bindings, and b, c, d, and e are the limits of functional projections, eliminations, imitations, and identifications, respectively. The values for a, b, c, d, and e are chosen following the intuition that functional projection and identification are the most explosive bindings. Figure 4.2 summarizes the effects of using different oracles.

The configuration of our procedure with no oracles outperforms the JP procedure with the first-order oracle. This suggests that the design of the procedure, in particular lazy normalization and lazy application of the substitution, already reduces the effects of the

[5] An archive with raw results, all used problems, and scripts for running each configuration is available at `http://doi.org/10.5281/zenodo.4269591`

	jp	cv	pv^{12}_{6666}	pv^{6}_{3333}	pv^{4}_{2222}	pv^{2}_{1222}	pv^{2}_{1121}	pv^{2}_{1020}
TPTP	1551	1717	1722	**1732**	**1732**	1715	1712	1719
SH	242	**260**	253	255	255	254	259	257

Figure 4.1: Proved problems, per configuration

	n	f	p	s	fp	fs	ps	fps
TPTP	1658	1717	1717	1720	1719	**1724**	1720	1723
SH	245	255	**260**	259	255	254	258	254

Figure 4.2: Proved problems, per used oracle

JP procedure's main bottlenecks. The raw evaluation data show that on TPTP benchmarks, complete and pragmatic configurations differ in the set of problems they solve—cv solves 19 problems not solved by pv^{4}_{2222}, whereas pv^{4}_{2222} solves 34 problems cv does not solve. Similarly, comparing the pragmatic configurations with each other, pv^{6}_{3333} and pv^{4}_{2222} each solve 13 problems that the other one does not. The overall higher success rate of pv^{2}_{1020} compared to pv^{2}_{1222} suggests that solving flex-flex pairs by trivial unifiers often suffices for superposition-based theorem proving.

In some cases, using oracles can hurt the performance of Zipperposition. Using oracles typically results in generating smaller CSUs, whose elements are more general substitutions than the ones we obtain without oracles. These more general substitutions usually contain more applied variables, which Zipperposition's heuristics avoid due to their explosive nature. This can make Zipperposition postpone necessary inferences for too long. Configuration n benefits from this effect and therefore solves 18 TPTP problems that no other configuration in Figure 4.2 solves. The same effect also gives configurations with only one oracle an advantage over configurations with multiple oracles on some problems.

The evaluation sheds some light on how often solid unification problems appear in practice. The raw data show that configuration s solves 5 TPTP problems that neither f nor p solve. Configuration f solves 8 TPTP problems that neither s nor p solve, while p solves 9 TPTP problems that two other configurations do not. This suggests that the solid oracle is slightly less beneficial than the fixpoint or pattern oracles, but still presents a useful addition to the set of available oracles.

A subset of TPTP benchmarks, concerning operations on Church numerals, is designed to test the efficiency of higher-order unification. Our procedure performs exceptionally well on these problems—it solves all of them, usually faster than other competitive higher-order provers. There are 11 benchmarks in the `NUM` category of TPTP that contain conjectures about Church numerals: `NUM020^1`, `NUM021^1`, `NUM415^1`, `NUM416^1`, `NUM417^1`, `NUM418^1`, `NUM419^1`, `NUM798^1`, `NUM799^1`, `NUM800^1`, and `NUM801^1`. We evaluated those problems using the same CPU nodes and the same time limits as above. In addition to Zipperposition, we used all higher-order provers that took part in the 2019 edition of CASC [160] (in the THF category) for this evaluation: CVC4 1.7 [12], Leo-III 1.4 [153], Satallax 3.4 [39], Vampire 4.4 THF [100]. Figure 4.3 shows the CPU time needed to solve a problem or "–" if the prover timed out.

	CVC4	Leo-III	Satallax	Vampire	Zipperposition (cv)
NUM020^1	–	0.46	–	–	**0.03**
NUM021^1	–	–	–	–	**4.10**
NUM415^1	45.80	0.34	0.21	0.42	**0.03**
NUM416^1	47.37	0.92	0.21	0.41	**0.07**
NUM417^1	–	49.73	**0.30**	0.40	0.45
NUM418^1	–	0.40	1.29	0.38	**0.03**
NUM419^1	–	0.42	23.33	0.37	**0.03**
NUM798^1	46.29	0.35	4.01	0.38	**0.03**
NUM799^1	–	5.05	–	–	**0.03**
NUM800^1	–	–	–	**0.37**	3.15
NUM801^1	–	0.73	38.77	–	**0.50**

Figure 4.3: Time needed to prove a problem, in seconds.

4.9 Discussion and Related Work

The problem addressed in this chapter is that of designing a complete and efficient higher-order unification procedure. Three main lines of research dominated the research field of higher-order unification over the last forty years.

The first line of research went in the direction of finding procedures that enumerate CSUs. The most prominent procedure designed for this purpose is the JP procedure [86]. Snyder and Gallier [150] also provide such a procedure, but instead of solving flex-flex pairs systematically, their procedure blindly guesses the head of the necessary binding by considering all constants in the signature and fresh variables of all possible types. Another approach, based on higher-order combinators, is given by Dougherty [56]. This approach blindly creates (partially applied) S-, K-, and I-combinator bindings for applied variables, which results in returning many redundant unifiers, as well as in nonterminating behavior even for simple problems such as $X\,\mathsf{a} = \mathsf{a}$.

The second line of research is concerned with enumerating preunifiers. The most prominent procedure in this line of research is Huet's [82]. The Snyder–Gallier procedure restricted to not solving flex-flex pairs is a version of the PT procedure presented in Sect. 4.5. It improves Huet's procedure by featuring a Solution rule.

The third line of research gives up the expressiveness of the full λ-calculus and focuses on decidable fragments. Patterns [124] are arguably the most important such fragment in practice, with implementations in Isabelle [125], Leo-III [153], Satallax [39], λProlog [118], and other systems. Functions-as-constructors [105] unification subsumes pattern unification but is significantly more complex to implement. Prehofer [133] lists many other decidable fragments, not only for unification but also preunification and unifier existence problems. Most of these algorithms are given for second-order terms with various constraints on their variables. Finally, one of the first decidability results is Farmer's discovery [61] that higher-order unification of terms with unary function symbols is decidable.

Our procedure draws inspiration from and contributes to all three lines of research. Accordingly, its advantages over previously known procedures can be laid out along those

three lines. First, our procedure mitigates many issues of the JP procedure. Second, it can be modified not to solve flex-flex pairs, and become a version of Huet's procedure with important built-in optimizations. Third, it can integrate any oracle for problems with finite CSUs, including the one we discovered.

The implementation of our procedure in Zipperposition was one of the reasons this prover evolved from proof-of-concept prover for higher-order logic to competitive higher-order prover. Shortly after the procedure was implemented, in the 2020 edition of CASC, Zipperposition won the THF division.

4.10 Conclusion

We presented a procedure for enumerating a complete set of higher-order unifiers that is designed for efficiency. Due to a design that restricts the search space and a tight integration of oracles, it reduces the number of redundant unifiers returned and gives up early in cases of nonunifiability. In addition, we presented a new fragment of higher-order terms that admits finite CSUs. Our evaluation shows a clear improvement over previously known procedures.

The procedure was a very prolific playground for tuning various heuristics to improve the success rate. In Chapter 6 we discuss some of the Zipperposition parameters we tuned to make better use of the procedure. The procedure also forms the basis of the unification procedure implemented in λE, the full higher-order extension of Ehoh, described in Chapter 7.

5

Boolean Reasoning in a Higher-Order Superposition Prover

Joint work with Visa Nummelin

We present a pragmatic approach to extending a Boolean-free higher-order superposition calculus to support Boolean reasoning. Our approach extends inference rules that have been used only in a first-order setting and uses some rules previously implemented in higher-order provers, as well as new rules. We have implemented the approach in the Zipperposition theorem prover. The evaluation shows highly competitive performance of our approach and a clear improvement over previous techniques.

In this work I implemented and evalauted most of the described approaches. Visa Nummelin implemented the FOOL preprocessing module.

5.1 Introduction

In Chapter 1 I motivated the work presented in this thesis by the need to bridge the gap between higher-order frontends and first-order backends. This gap is traditionally bridged using translations from higher-order to first-order logic [117, 141]. However, as shown in Chapter 3, translations are usually less efficient than native support. The distinguishing features of higher-order logic used by proof assistants that the translation must eliminate include λ binders, function extensionality—the property that functions are equal if they agree on every argument, described by the axiom $\mathbf{\forall} x\, y.\, (\mathbf{\forall} z.\, x\, z \approx y\, z) \longrightarrow x \approx y$—and formulas occurring as arguments of function symbols [117].

As mentioned in Sect. 2.5.6, Bentkamp et al. developed λSup, a complete higher-order calculus that alleviates the need to translate λ-terms and function extensionality. Kotelnikov et al. [98, 99] extended the language of first-order logic to support the third feature of higher-order logic that requires translation. They described two approaches: One based on a calculus-level treatment of Booleans and another that requires no changes to the calculus, based on preprocessing.

To fully bridge the gap between higher-order and first-order tools, we combine the two approaches: We take λSup as our basis and extend it with inference rules that reason with Boolean terms. In early work, Kotelnikov et al. [99] have described a *FOOL paramodulation* rule that, under some order requirements, removes the need for the axiom describing the Boolean domain—$\mathbf{\forall} p.\ p \approx \top \vee p \approx \bot$. In this approach, it is assumed that a problem with formulas occurring as arguments of symbols is translated to first-order logic using a translation they describe.

The backbone of our approach is based on an extension of this rule to higher-order logic: We do not translate away any Boolean structure that is nested inside non-Boolean terms and allow our rule to hoist the nested Booleans to the literal level. Then, we clausify the resulting formula (i.e., a clause containing formulas in literals) using a new rule.

An important feature that we inherit by building on top of λSup is support for function extensionality. Moving to higher-order logic with Booleans also means that we need to consider *Boolean extensionality*: $\mathbf{\forall} p\, q.\, (p \leftrightarrow q) \longrightarrow p \approx q$. We extend the rules of λSup that treat function extensionality to also treat Boolean extensionality.

Rules that extend the two orthogonal approaches form the basis of our support for Boolean reasoning (Sect. 5.3). In addition, we have implemented rules that are inspired by the ones used in the higher-order provers Leo-III [153] and Satallax [39], such as elimination of Leibniz equality, primitive instantiation, and treatment of the choice operator [2]. We have also designed new rules including those that use higher-order unification to resolve Boolean formulas that are hoisted to the literal level, delay clausification of nonatomic literals, and reason about formulas under λ-binders. Even though the rules are inspired by the ones of refutationally complete higher-order provers, we do not guarantee completeness of our extension of λSup. Using the insights that we gained with this incomplete extension of λSup to full higher-order logic, Bentkamp et al. designed a complete calculus for full higher-order logic—$o\lambda$Sup [17].

We compare our native approach with two alternatives that are based on preprocessing (Sect. 5.4). First, we compare it to an axiomatization of the theory of Booleans. Second, inspired by work of Kotelnikov et al. [98], we implement the preprocessing approach that does not require introduction of Boolean axioms. We discuss some examples, coming from

TPTP [157], that illustrate advantages and disadvantages of our approach (Sect. 5.5).

The theorem prover Zipperposition [48, 49] was used by Bentkamp et al. to implement λSup. We further extend their implementation based on the approach presented in this chapter.

We performed an extensive evaluation of our approach (Sect. 5.6). In addition to evaluating different configurations of our new rules, we have compared them to full higher-order provers CVC4, Leo-III, Satallax, and Vampire. The results suggest that it is beneficial to natively support Boolean reasoning—the approach outperforms preprocessing-based approaches. Furthermore, it is very competitive in comparison to state-of-the-art higher-order provers. We discuss the differences between our approach and the approaches our work is based on, as well as related approaches (Sect. 5.7).

5.2 Background

We base our work (and parts of the following text) on Bentkamp et al.'s [18] extensional polymorphic clausal higher-order logic. This logic supports rank-1 polymorphism, as defined by the TH1 format of the TPTP [88]. We extend the syntax of this logic by adding logical connectives to the signature. The semantics of the logic is extended by interpreting Boolean type o as a two-element domain. This amounts to extending the logic to full higher-order logic. Taking a different perspective, this logic is an extension of the one presented in Chapter 3 with λ-abstraction, polymorphic types, and native logical connectives. For reference, we provide a description of this polymorphic logic. For the notions that are not defined here, we assume the definitions introduced in Chapter 2.

A *signature* is a quadruple $(\Sigma_{\mathsf{ty}}, \mathscr{V}_{\mathsf{ty}}, \Sigma, \mathscr{V})$ where Σ_{ty} is a set of type constructors, $\mathscr{V}_{\mathsf{ty}}$ is a set of type variables and Σ and $\mathscr{V}$ are sets of constants and term variables, respectively. We require nullary type constructor o as well as binary constructor $\to$ to be in Σ_{ty}. Types τ, υ are either type variables $\alpha \in \mathscr{V}_{\mathsf{ty}}$ or of the form $\kappa(\tau_1, \ldots \tau_n)$ where κ is an n-ary type constructor. We write κ for $\kappa()$, $\tau \to \upsilon$ for $\to(\tau, \upsilon)$, and we drop parentheses to shorten $\tau_1 \to (\cdots \to (\tau_{n-1} \to \tau_n)\cdots)$ into $\tau_1 \to \cdots \to \tau_n$. Each symbol in Σ is assigned a type declaration of the form $\Pi\overline{\alpha}_n.\ \tau$ where all variables occurring in τ are among $\overline{\alpha}_n$.

Function symbols $\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, \ldots$ are elements of Σ; their type declarations are written as $\mathsf{f} : \Pi\overline{\alpha}_n.\ \tau$. Set $\mathscr{V}$ contains both free and bound variables, following the distinction introduced in Sect. 2.3. Free term variables from the set $\mathscr{V}$ are written $F, G, X, Y, \ldots$ and we denote their types as $X : \tau$. Bound term variables are written $x, y, z, \ldots$, and their types are similarly denoted. When the type is not important, we omit type declarations. We assume that symbols $\top, \bot, \neg, \wedge, \vee, \to, \leftrightarrow$ with their standard meanings and type declarations are elements of Σ. Furthermore, we assume that polymorphic symbols $\forall$ and $\exists$ with type declarations $\Pi\alpha.\ (\alpha \to o) \to o$ and $\approx : \Pi\alpha.\ \alpha \to \alpha \to o$ are in Σ, with their standard meanings. All these symbols are called *logical symbols*. We use infix notation for binary logical symbols.

Terms are defined inductively as follows. Free $(X : \tau)$ and bound $(x : \tau)$ variables are terms of type τ. If $\mathsf{f} : \Pi\overline{\alpha}_n.\ \tau$ is in Σ and $\overline{\upsilon}_n$ is a tuple of types, called type arguments, then $\mathsf{f}\langle\overline{\upsilon}_n\rangle$ (written as f if $n = 0$, or if type arguments can be inferred from the context) is a term of type $\tau\{\overline{\alpha}_n \mapsto \overline{\upsilon}_n\}$, called constant. If x is a bound variable of type τ and s is a term of type υ, then $\lambda x.\ s$ is a term of type $\tau \to \upsilon$. If s and t are of type $\tau \to \upsilon$ and τ, respectively, then $s\,t$ is a term of type υ. We call terms of Boolean type (o) *formulas* and denote them by $f, g, h, \ldots$; we use $P, Q, R, \ldots$ for free variables whose result type is o and $\mathsf{p}, \mathsf{q}, \mathsf{r}$ for

constants with the same result type. Formulas whose top-level symbol is not logical are called *atoms*. Unless stated otherwise, we view terms as $\alpha\beta\eta$-equivalence classes, with the η-long β-reduced form as the representative.

Given a formula f, we call its Boolean subterm $f|_p$ a *top-level Boolean* if for all proper prefixes q of p, the head of $f|_q$ is a logical constant. Otherwise, we call it a *nested Boolean*. For example, in the formula $f = \mathsf{h}\,\mathsf{a} \approx \mathsf{g}(\mathsf{p} \longrightarrow \mathsf{q}) \vee \neg \mathsf{p}$, $f|_1$ and $f|_2$ are top-level Booleans, while $f|_{1.2.1}$ is a nested Boolean, as well as its subterms. First-order logic allows only top-level Booleans, whereas nested Booleans are characteristic for higher-order logic.

Our calculus works with the clausal structure described in Sect. 2.4. However, its higher-order nature allows representing formula f as a singleton clause containing only literal f.

5.3 The Native Approach

Zipperposition already had some support for Boolean reasoning before we started extending λSup. In this section, we first describe the internals of Zipperposition responsible for reasoning with Booleans. We continue by describing the rules that we have implemented. For ease of presentation we divide them into three categories.

5.3.1 Support for Booleans in Zipperposition

As mentioned in Chapter 4, Zipperposition is an open source prover written in OCaml. From its inception, it was designed as a prover that supports easy extension of its base superposition calculus to various theories, including arithmetic, induction, and limited support for higher-order logic [48, 49].

In Zipperposition, applications are represented in flattened, spine notation (Sect. 3.3). This means that the higher-order term $(\mathsf{f}\,\mathsf{a})\,\mathsf{b}$ is represented the same as the first-order term $\mathsf{f}(\mathsf{a},\mathsf{b})$. In addition, Zipperposition uses associativity of $\wedge$ and $\vee$ to flatten out the nested applications of these symbols. For example, terms $\mathsf{p} \wedge (\mathsf{q} \wedge \mathsf{r})$ and $(\mathsf{p} \wedge \mathsf{q}) \wedge \mathsf{r}$ are internally stored as $\wedge\,\mathsf{p}\,\mathsf{q}\,\mathsf{r}$. Zipperposition's support for λ-terms is used to represent quantified nested Booleans: Formulas $\forall x.\ f$ and $\exists x.\ f$ are represented as $\forall(\lambda x.\ f)$ and $\exists(\lambda x.\ f)$. After clausification of the input problem, no nested Booleans are modified or renamed using fresh predicate symbols.

The version of Zipperposition preceding our modifications distinguished between nonequational and equational literals. Following E [147], we modified Zipperposition to represent all literals equationally: a nonequational literal f is stored as $f \approx \top$, whereas $\neg f$ is stored as $f \not\approx \top$. Equations of the form $f \approx \bot$ and $f \not\approx \bot$ are transformed into $f \not\approx \top$ and $f \approx \top$, respectively.

5.3.2 Core Rules

Kotelnikov et al. [99], to the best of our knowledge, pioneered the approach of extending a first-order superposition prover to support nested Booleans. They call including the axiom $\forall p.\ p \approx \top \vee p \approx \bot$ "a recipe for disaster". To combat the explosive behavior of the axiom, they impose the following two requirements to the simplification order $\succ$: $\top \succ \bot$ and $\top$ and $\bot$ are two smallest ground terms with respect to $\succ$. If these requirements are met, there is no self-paramodulation of the clause and the only paramodulation possible is from the

literal $p \approx \mathsf{T}$ of the mentioned axiom into a Boolean subterm of another clause. Finally, Kotelnikov et al. replace the axiom with the inference rule *FOOL paramodulation* (FP):

$$\frac{C[f]}{C[\mathsf{T}] \vee f \approx \bot}\,\text{FP}$$

where f is a nested non-variable Boolean subterm of clause C, different from T and $\bot$. In addition, they translate the initial problem containing nested Booleans to first-order logic without interpreted Booleans; thus, the symbols T and $\bot$ and the type o correspond to proxy symbols and types introduced during the translation.

We created two rules that are syntactically similar to FP but are adapted for higher-order logic with one key distinction—we do not perform any translation:

$$\frac{C[f]}{C[\bot] \vee f \approx \mathsf{T}}\,\textsc{Cases} \qquad \frac{\begin{array}{c}C[f]\\\hline\hline\end{array}}{C[\bot] \vee f \approx \mathsf{T} \quad C[\mathsf{T}] \vee f \not\approx \mathsf{T}}\,\textsc{CasesSimp}$$

The prover that uses the rules should not include them both at the same time. In addition, since literals $f \approx \bot$ are represented as negative equations $f \not\approx \mathsf{T}$, which cannot be used to paramodulate from, we change the first requirement on the order to $\bot \succ \mathsf{T}$, making T the smallest symbol.

These two rules hoist Boolean subterms f to the literal level; therefore, some conclusions of Cases and CasesSimp will have literals of the form $f \approx \mathsf{T}$ (or $f \not\approx \mathsf{T}$) where f is not an atom. This introduces the need for the rule called immediate clausification (IC), where $\overline{D}_m$ are defined below:

$$\frac{\begin{array}{c}C\\\hline\hline\end{array}}{D_1 \;\cdots\; D_m}\,\text{IC}$$

Let $s \mathrel{\dot{\approx}} t$ denote $\approx$ or $\not\approx$. We say that a clause is *standard* if all of its literals are of the form $s \mathrel{\dot{\approx}} t$, where s and t are not Booleans or of the form $f \mathrel{\dot{\approx}} \mathsf{T}$, where the head of f is not a logical symbol. The rule IC is applicable if clause $C = L_1 \vee \cdots \vee L_n$ is not standard. The resulting clauses $\overline{D}_m$ represent the result of clausification of the formula $\forall \overline{x}.\ L_1 \vee \cdots \vee L_n$ where $\overline{x}$ consists of all free variables of C. Using Boolean extensionality, Zipperposition's clausification algorithm treats Boolean equality as equivalence (i.e., it replaces $\approx\langle o\rangle$ with $\leftrightarrow$).

An advantage of leaving nested Booleans unmodified is that the prover will be able to prove some problems containing them without using the prolific rules described above. For example, given two clauses $\mathsf{f}(\mathsf{p}\,X \longrightarrow \mathsf{p}\,Y) \approx \mathsf{a}$ and $\mathsf{f}(\mathsf{p}\,\mathsf{a} \longrightarrow \mathsf{p}\,\mathsf{b}) \not\approx \mathsf{a}$, the empty clause can easily be derived without the above rules. A disadvantage of this approach is that the proving process will periodically be interrupted by expensive calls to the clausifier.

If implemented naively, rules Cases and CasesSimp can result in many redundant clauses. Consider the following example: let $\mathsf{p} : o \longrightarrow o$, $\mathsf{a} : o$ and consider a clause set containing $\mathsf{p}(\mathsf{p}(\mathsf{p}\,\mathsf{a})) \approx \mathsf{T}$. Then, the clause $C = \mathsf{a} \approx \mathsf{T} \vee \mathsf{p}\bot \approx \mathsf{T}$ can be derived in eight ways using the rules, depending on which nested Boolean subterm was chosen for the inference. In general, if a clause has a subterm occurrence of the form $\mathsf{p}^n\,\mathsf{a}$, where both p and a have result type o, the clause $\mathsf{a} \approx \mathsf{T} \vee \mathsf{p}\bot \approx \mathsf{T}$ can be derived in at least 2^{n-1} ways. To combat these issues we implemented pragmatic restrictions of the rule: only the f which is

the leftmost outermost (or innermost) eligible subterm will be considered. With this modification C can be derived in only one way. Furthermore, some intermediate conclusions of the rules will not be derived, pruning the search space.

The clausification algorithm by Nonnengart and Weidenbach [126] eagerly simplifies the input problem using Boolean equivalences before clausifying it. For example, the formula $\mathsf{p} \wedge \top$ is replaced by p. To simplify nested Booleans we implemented the rule BoolSimp parameterized by a set of rewrite rules E:

$$\frac{C[\sigma(f)]}{C[\sigma(g)]}\,\textsc{BoolSimp}$$

where $f \longrightarrow g \in E$ and σ is any substitution. In the current implementation of Zipperposition, E contains the rules described by Nonnengart and Weidenbach [126, Sect. 3]. This set contains the rules describing how each logical symbol behaves when given arguments $\top$ or $\bot$: for example, it includes $(\top \longrightarrow p) \longrightarrow p$ and $(p \longrightarrow \top) \longrightarrow \top$. Leo-III implements a similar rule, called simp [151, Sect. 4.2.1.].

Our decision to represent negative atoms as negative equations was motivated by the need to alter Zipperposition's earlier behavior as little as possible. Namely, negative atoms are not used as literals that can be used to paramodulate from, and as such added to the laziness of the superposition calculus. However, it might be useful to consider unit clauses of the form $f \not\approx \top$ as $f \approx \bot$ to strengthen rewriting. To that end, we have introduced the following rule:

$$\frac{f \not\approx \top \qquad C[\sigma(f)]}{f \not\approx \top \qquad C[\bot]}\,\textsc{BoolDemod}$$

5.3.3 Higher-Order Considerations

To achieve refutational completeness of higher-order resolution and similar calculi, it is necessary to instantiate variables with result type o, *predicate variables*, with arbitrary formulas [2, 151]. Fortunately, we can approximate the formulas using a complete set of logical symbols (e.g., $\neg$, $\forall$, and $\wedge$). Since such an approximation is not only necessary for completeness of some calculi, but possibly useful in practice, we implemented the *primitive instantiation* (PI) rule:

$$\frac{C \vee (\lambda \overline{x}.\, P\, \overline{s}_n) \mathrel{\dot{\approx}} t}{\{P \mapsto f\}(C \vee (\lambda \overline{x}.\, P\, \overline{s}_n) \mathrel{\dot{\approx}} t)}\,\text{PI}$$

where P is a free variable of the type $\tau_1 \longrightarrow \cdots \longrightarrow \tau_n \longrightarrow o$. Choosing a different f that instantiates P, we can balance between explosiveness of approximating a complete set of logical symbols and incompleteness of pragmatic approaches. We borrow the notion of imitation from higher-order unification jargon (Sect. 4.3), and we say that the term $\lambda \overline{x}_m.\, \mathsf{f}(Y_1\, \overline{x}_m) \cdots (Y_n\, \overline{x}_m)$ is an *imitation* of constant $\mathsf{f} : \tau_1 \longrightarrow \cdots \longrightarrow \tau_n \longrightarrow \tau$ for some variable Z of type $\nu_1 \longrightarrow \cdots \longrightarrow \nu_m \longrightarrow \tau$. Variables $\overline{Y}_n$ are fresh free variables, where each Y_i has the type $\nu_1 \longrightarrow \cdots \longrightarrow \nu_m \longrightarrow \tau_i$; variable x_i is of type ν_i.

Rule PI was already implemented by Simon Cruanes in Zipperposition, before we started our modifications. Its implementation contains the following modes that generate sets of possible terms f for $P : \tau_1 \longrightarrow \cdots \longrightarrow \tau_n \longrightarrow o$: *Full*, *Pragmatic*, and $Imit_\star$. In $Imit_\star$,

$\star$ is an element of a set of logical constants $S = \{\wedge, \vee, \approx\langle\alpha\rangle, \neg, \forall, \exists\}$. Mode *Full* generates imitations (for P) of all elements of S. Mode *Pragmatic* generates imitations of $\neg$, $\top$, and $\bot$; if there exist indices i, j such that $i \neq j$ and $\tau_i = \tau_j$, then it generates $\lambda\overline{x}_n.\, x_i \approx x_j$; if there exist indices i, j such that $i \neq j$, and $\tau_i = \tau_j = o$, then it generates $\lambda\overline{x}_n.\, x_i \wedge x_j$ and $\lambda\overline{x}_n.\, x_i \vee x_j$; if for some i, $\tau_i = o$, then it generates $\lambda\overline{x}_n.\, x_i$. Mode $Imit_\star$ generates imitations of $\top$, $\bot$, and $\star$. In addition, $Imit_{\forall\exists}$ generates imitations of both $\forall$ and $\exists$.

While experimenting with our implementation we noticed some proof patterns that led us to come up with the following modifications. First, it often suffices to perform PI only on initial clauses—which is why we allow the rule to be applied only to the clauses created using at most k generating inferences. Second, if the rule was used in the proof, its premise is usually only used as part of that inference—which is why we implemented a version of PI that removes the clause after all possible PI inferences have been performed. We observed that the mode $Imit_\star$ is useful in practice since often approximation of a single logical symbol suffices.

The axiom of choice is notoriously difficult to handle efficiently in higher-order provers. Andrews formulates this axiom as $\forall p.\, (\exists x.\, p\, x) \longrightarrow p\, (\varepsilon\, p)$, where $\varepsilon : \Pi\alpha.\, (\alpha \longrightarrow o) \longrightarrow \alpha$ denotes the *choice operator* [2]. After clausification, this axiom becomes $P\, X \not\approx \top \vee P\, (\varepsilon\, P) \approx \top$. Since the term $P\, X$ matches any Boolean term in the proof state, this axiom is very explosive. Therefore, Leo-III [153] heuristically recognizes symbols that correspond to choice and deals with them on the calculus level. Namely, whenever a clause $C = P\, X \not\approx \top \vee P\, (\mathsf{f}\, P) \approx \top$ is chosen for processing, C is removed from the proof state and f is added to the set of choice functions CF (which initially contains just ε). Later, elements of CF are used to heuristically instantiate the axiom of choice. We reused the method of recognizing choice functions, but generalized the rule for creating the instance of the axiom (assuming $\xi \in CF$ and X and z are fresh variables):

$$\frac{C[\xi\, t]}{X\, (t\, Y) \not\approx \top \vee X\, (t\, (\xi\, (\lambda z.\, X\, (t\, z)))) \approx \top}\ \textsc{Choice}$$

Let D be the conclusion of Choice. The fresh variable X in D acts as an arbitrary context around t, the chosen instantiation for P from the axiom of choice; the variable X can later be replaced by imitations of logical symbols to create more complex instantiations of the choice axiom. To generate useful instances early, we create two instances: $\{X \mapsto \lambda z.\, z\}(D)$ and $\{X \mapsto \lambda z.\, \neg\, z\}(D)$. Then, depending on Zipperposition options, D will either be deleted or kept. Note that D will not subsume its instances, as the matching algorithm Zipperposition uses favors performance over completeness and is thus too weak for this [18, Sect. 6]

Most provers natively support both functional and Boolean extensionality reasoning: Bhayat et al. [25] modify first-order unification to return unification constraints consisting of pairs of terms of functional type, whereas Steen relies on the unification rules of Leo-III's calculus [151, Sect. 4.3.3] to deal with extensionality. As a pragmatic extension of the λSup calculus, Bentkamp et al. [18] propose to alter the core generating inference rules of superposition (Sect. 2.5.3) to support functional extensionality. Instead of requiring that terms involved in the inference are unifiable, it is required that they can be decomposed into *disagreement pairs* such that at least one of the disagreement pairs is of functional type. Disagreement pairs of terms s and t of the same type are defined inductively using function dp: $\mathsf{dp}(s, t) = \emptyset$ if s and t are equal; $\mathsf{dp}(a\, \overline{s}_n, b\, \overline{t}_m) = \{(a\, \overline{s}_n, b\, \overline{t}_m)\}$ if a and b are different heads;

5

$\mathsf{dp}(\lambda x.\, s, \lambda y.\, t) = \{(\lambda x.\, s, \lambda y.\, t)\}$; $\mathsf{dp}(a\,\overline{s}_n, a\,\overline{t}_n) = \bigcup_{i=1}^{n} \mathsf{dp}(s_i, t_i)$. Then the extensionality rules are stated as follows:

$$\frac{s \approx t \vee C \qquad u[s'] \mathrel{\dot{\approx}} v \vee D}{\sigma(s_1 \not\approx s'_1 \vee \cdots \vee s_n \not\approx s'_n \vee u[t] \mathrel{\dot{\approx}} v \vee C \vee D)}\ \textsc{AbsSup}$$

$$\frac{s \not\approx s' \vee C}{\sigma(s_1 \not\approx s'_1 \vee \cdots \vee s_n \not\approx s'_n \cdots \vee C)}\ \textsc{AbsER}$$

$$\frac{s \approx t \vee s' \approx u \vee C}{\sigma(s_1 \not\approx s'_1 \vee \cdots \vee s_n \not\approx s'_n \vee t \not\approx u \vee s' \approx u \vee C)}\ \textsc{AbsEF}$$

In each of the rules, σ is an MGU of the types of s and s', and $\mathsf{dp}(\sigma(s), \sigma(s')) = \{(s_1, s'_1), \ldots, (s_n, s'_n)\}$. Rules AbsSup, AbsER, and AbsEF are extensional versions of superposition, equality resolution, and equality factoring (Sect. 2.5.3). All side conditions for extensional rules are the same as for the standard rules, except that the condition that s and s' are unifiable is replaced by the condition that $n > 0$ and at least one s_i is of functional type. By Abs we denote the union of these three rules. We extend Abs to support Boolean extensionality by requiring that at least one s_i is of functional or type o, and adding the condition "$\mathsf{dp}(f, g) = \{(f, g)\}$ if f and g are different formulas" to the definition of dp. If another condition of dp's definition is also applicable, the newly added one is preferred. In the rest of the text Abs refers to the version that supports Boolean extensionality.

Consider the clause $\mathsf{f}(\neg \mathsf{p} \vee \neg \mathsf{q}) \not\approx \mathsf{f}(\neg(\mathsf{p} \wedge \mathsf{q}))$. This clause is unsatisfiable, as the arguments of f on the different sides of the disequation are Boolean-extensionally equal. Without the Abs rules Zipperposition relies on the Cases(Simp) and IC rules to derive the empty clause. On the other hand, AbsER generates $C = \neg \mathsf{p} \vee \neg \mathsf{q} \not\approx \neg(\mathsf{p} \wedge \mathsf{q})$. Then, C gets clausified using IC, effectively reducing the problem to $\neg(\neg \mathsf{p} \vee \neg \mathsf{q} \leftrightarrow \neg(\mathsf{p} \wedge \mathsf{q}))$, which is first-order.

Zipperposition restricts AbsSup by requiring that s and s' are not of function or Boolean type. If the terms are of function type, our experience is that a better treatment of function extensionality is to apply fresh free variables (or Skolem terms, depending on the sign [18]) to both sides of a (dis)equation to reduce it to a first-order literal; Boolean extensionality is usually better supported by applying IC on the top-level Boolean term. Thus, for the following discussion, we assume s and s' are not λ-abstractions or formulas. Then, AbsSup is applicable if s and s' have the same head, and a functional or Boolean subterm. To speed up retrieval of such terms, we added an index that maps symbols to positions in clauses where they appear as a head of a term that has a functional or Boolean subterm. This index is empty for first-order problems, incurring no overhead if extensionality reasoning is not needed. One more restriction we implemented is that we do not apply the Abs rules if all disagreement pairs have at least one side whose head is a variable; those will be dealt with more efficiently using the core rules of the superposition calculus. To simplify the resulting clauses of Abs rules, we also eagerly remove literals $s_i \not\approx s'_i$ using a simplifying version of ER (Sect. 2.5.3). In particular, we apply the first unifier returned by the terminating, pragmatic variant of unification algorithm described in Sect. 4.3, and remove the premise after the conclusion of ER has been computed.

Expressiveness of higher-order logic allows users to define equality using a single axiom, called Leibniz equality [2]: $\forall xy.(\forall p.\, p\,x \longrightarrow p\,y) \longrightarrow x \approx y$. Intuitively, it embodies

the principle that if two terms are indistinguishable by propositions, they must be equal. Leibniz equality often appears in TPTP problems [157]. Since modern provers have native support for equality, it is usually beneficial to recognize and replace occurrences of Leibniz equality by the natively supported one.

Before we did our modifications, Zipperposition had a powerful rule that recognizes clauses that contain variations of Leibniz equality and instantiates them with native equality. This rule was designed by Simon Cruanes, and to the best of our knowledge, it has not been documented so far. With his permission we describe this rule as follows:

$$\frac{P\,\overline{s}_n^1 \approx \mathsf{T} \vee \cdots \vee P\,\overline{s}_n^i \approx \mathsf{T} \vee P\,\overline{t}_n^1 \not\approx \mathsf{T} \vee \cdots \vee P\,\overline{t}_n^j \not\approx \mathsf{T} \vee C}{\sigma(P\,\overline{s}_n^1 \approx \mathsf{T} \vee \cdots \vee P\,\overline{s}_n^i \approx \mathsf{T}) \vee C}\ \textsc{ElimPredVar}$$

where P is a free variable, that does not occur in any s_k^l or t_k^l, or in C; σ is defined as $\{P \mapsto \lambda\overline{x}_n.\bigvee_{k=1}^{j}(\bigwedge_{l=1}^{n} x_l \approx t_l^k)\}$.

To better understand how this rule removes variable-headed negative literals, consider the clause $C = P\,\mathsf{a}_1\,\mathsf{a}_2 \approx \mathsf{T} \vee P\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \mathsf{T} \vee P\,\mathsf{c}_1\,\mathsf{c}_2 \not\approx \mathsf{T}$. The rule ElimPredVar will generate $\sigma = \{P \mapsto \lambda xy.(x \approx \mathsf{b}_1 \wedge y \approx \mathsf{b}_2) \vee (x \approx \mathsf{c}_1 \wedge y \approx \mathsf{c}_2)\}$. After applying σ to C and subsequent β-reduction, the negative literal $P\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \mathsf{T}$ will reduce to $(\mathsf{b}_1 \approx \mathsf{b}_1 \wedge \mathsf{b}_2 \approx \mathsf{b}_2) \vee (\mathsf{b}_1 \approx \mathsf{c}_1 \wedge \mathsf{b}_2 \approx \mathsf{c}_2) \not\approx \mathsf{T}$, which is equivalent to $\bot$. Thus, we can remove this literal and all negative literals of the form $P\,\overline{t}_n \not\approx \mathsf{T}$ from C and apply σ to the remaining ones.

The previous rule removes all variables occurring in disequations in one attempt. We implemented two rules that behave more lazily, inspired by the ones present in Leo-III and Satallax:

$$\frac{P\,\overline{s}_n \approx \mathsf{T} \vee P\,\overline{t}_n \not\approx \mathsf{T} \vee C}{\sigma(s_i \approx t_i \vee C)}\ \textsc{ElimLeibniz+} \qquad \frac{P\,\overline{s}_n \not\approx \mathsf{T} \vee P\,\overline{t}_n \approx \mathsf{T} \vee C}{\sigma'(s_i \approx t_i \vee C)}\ \textsc{ElimLeibniz−}$$

where P is a free variable that does not occur in t_i, $\sigma = \{P \mapsto \lambda\overline{x}_n.\, x_i \approx t_i\}$, and $\sigma' = \{P \mapsto \lambda\overline{x}_n.\neg(x_i \approx t_i)\}$. ElimLeibniz directly applies Leibniz equality: When the premise of Leibniz equality is clausified it becomes $P\,X \not\approx \mathsf{T} \vee P\,Y \approx \mathsf{T}$. The rule then tries to find a similar two-literal subclause in a clause C, and if successful instantiates C with a substitution that asserts equality of X and Y.

This rule differs from ElimPredVar in three ways. First, it can replace a predicate variable both with equality and disequality. Second, due to its simplicity, it usually does not require IC as the following step. Third, it imposes much weaker conditions on P. However, removing all negative variables in one step might improve performance by generating fewer intermediate clauses. Coming back to the example of the clause $C = P\,\mathsf{a}_1\,\mathsf{a}_2 \approx \mathsf{T} \vee P\,\mathsf{b}_1\,\mathsf{b}_2 \not\approx \mathsf{T} \vee P\,\mathsf{c}_1\,\mathsf{c}_2 \not\approx \mathsf{T}$, we can apply ElimLeibniz+ using the substitution $\sigma = \{P \mapsto \lambda xy.\, x \approx \mathsf{b}_1\}$ to obtain the clause $C' = \mathsf{a}_1 \approx \mathsf{b}_1 \vee \mathsf{a}_1 \not\approx \mathsf{c}_1$, which is considerably simpler than the one obtained by ElimPredVar.

5.3.4 Additional Rules

Zipperposition's pragmatic, incomplete unification algorithm uses a flattened representation of terms with logical operators $\wedge$ and $\vee$ as heads to unify terms that are not unifiable modulo $\alpha\beta\eta$-equivalence, but are unifiable modulo associativity and commutativity of $\wedge$

and ∨. Let ◇ denote either ∧ or ∨. When the unification algorithm is given two terms $\diamond\,\overline{s}_n$ and $\diamond\,\overline{t}_n$, where neither of $\overline{s}_n$ nor $\overline{t}_n$ contains duplicates, it performs the following steps: First, it removes all terms that appear in both $\overline{s}_n$ and $\overline{t}_n$ from the two argument tuples. Next, the remaining terms are sorted first by their head term and then their syntactic weight. Finally, an attempt is made to unify sorted lists pairwise. As an example, consider the problem of unifying the pair $\wedge\,(\mathsf{p}\,\mathsf{a})\,(\mathsf{q}\,(\mathsf{f}\,\mathsf{a}))$ $\stackrel{?}{=}$ $\wedge\,(\mathsf{q}\,(\mathsf{f}\,\mathsf{a}))\,(R\,(\mathsf{f}\,(\mathsf{f}\,\mathsf{a})))$ where R is a free variable. If the arguments of ∧ are simply sorted as described above, we would try to unify $\mathsf{p}\,\mathsf{a}$ with $\mathsf{q}\,(\mathsf{f}\,\mathsf{a})$, and fail to find a unifier. However, by removing the term $\mathsf{q}\,(\mathsf{f}\,\mathsf{a})$ from the argument lists, we will be left with the problem $\mathsf{p}\,\mathsf{a} \stackrel{?}{=} R\,(\mathsf{f}\,(\mathsf{f}\,\mathsf{a}))$ which has a unifier. This approach enables us to find more unifiers than by simple syntactic unification.

The winner of the higher-order theorem division of the 2019 edition of CASC [160], Satallax [39], has one crucial advantage over Zipperposition: it is based on higher-order tableaux, and as such it does not require formulas to be converted to clauses. The advantage of tableaux is that once it instantiates a variable with a term, this instantiation naturally propagates through the whole formula. In Zipperposition, which is based on λSup, the original formula is clausified and instantiating a variable in a clause C does not automatically instantiate it in all clauses that are results of clausification of the same formula as C. To mitigate this issue, we have created extensions of equality resolution and equality factoring that take Boolean extensionality into account:

$$\frac{s \approx s' \vee C}{\sigma(C)}\,\textsc{BoolER} \qquad \frac{P\,\overline{s}_n \approx \mathsf{T} \vee s' \not\approx \mathsf{T} \vee C}{\sigma(s' \not\approx \mathsf{T} \vee C)}\,\textsc{BoolEF+}-$$

$$\frac{P\,\overline{s}_n \not\approx \mathsf{T} \vee s' \approx \mathsf{T} \vee C}{\sigma(s' \approx \mathsf{T} \vee C)}\,\textsc{BoolEF}-+ \qquad \frac{P\,\overline{s}_n \not\approx \mathsf{T} \vee s' \not\approx \mathsf{T} \vee C}{\sigma(s' \not\approx \mathsf{T} \vee C)}\,\textsc{BoolEF}--$$

All side conditions except for the ones concerning the unifiability of terms are as in the original equality resolution and equality factoring rules. In rule BoolER, σ is a unifier of s and $\neg\, s'$. In the $+-$ and $-+$ versions of BoolEF, σ unifies $P\,\overline{s}_n$ and $\neg\, s'$, and in the remaining version it unifies $P\,\overline{s}_n$ and s'. Intuitively, these rules bring Boolean (dis)equations in the appropriate form for application of the corresponding base rules. In particular, for BoolER we interpret the equation $s \approx s'$ as $s \not\approx \neg\, s'$, which allows us to simulate ER inference from Sect. 2.5.3. For the BoolEF family of rules, we convert disequations into equations using the trick of negating one side to simulate EF. The rule BoolER only considers literals of the form $s \approx t$. Equivalences $s \leftrightarrow t = \mathsf{T}$ and disequivalences $(s \leftrightarrow t) \not\approx \mathsf{T}$ are automatically simplified to $s \approx t$ or $s \not\approx t$. The example `SET557^1` in Sect. 5.5 illustrates how the Bool family of rules helps solve problems that get obfuscated by clausification.

Another approach to mitigate harmful effects of immediate clausification is to delay it as long as possible. Following the approach by Ganzinger and Stuber [69], we represent every input formula f as a unit clause $f \approx \mathsf{T}$ and use the following delayed clausification (DC) rules:

$$\frac{(g \wedge h) \approx \mathsf{T} \vee C}{g \approx \mathsf{T} \vee C \quad h \approx \mathsf{T} \vee C}\,\mathrm{DC}_{\wedge} \qquad \frac{(g \vee h) \approx \mathsf{T} \vee C}{g \approx \mathsf{T} \vee h \approx \mathsf{T} \vee C}\,\mathrm{DC}_{\vee} \qquad \frac{(g \rightarrow h) \approx \mathsf{T} \vee C}{g \not\approx \mathsf{T} \vee h \approx \mathsf{T} \vee C}\,\mathrm{DC}_{\rightarrow}$$

$$\frac{(\neg g) \approx \mathsf{T} \vee C}{g \not\approx \mathsf{T} \vee C}\,\mathrm{DC}_{\neg} \qquad \frac{(\forall x.\, g) \approx \mathsf{T} \vee C}{\{x \mapsto Y\}(g) \vee C}\,\mathrm{DC}_{\forall} \qquad \frac{(\exists x.\, g) \approx \mathsf{T} \vee C}{\{x \mapsto \mathsf{sk}\langle \overline{\alpha} \rangle\, \overline{Y}_n\}(g) \vee C}\,\mathrm{DC}_{\exists}$$

$$\frac{g \approx h \vee C}{g \not\approx \mathsf{T} \vee h \approx \mathsf{T} \vee C \quad g \approx \mathsf{T} \vee h \not\approx \mathsf{T} \vee C}\,\mathrm{DC}_{\approx}$$

In $\mathrm{DC}_{\approx}$ we require both g and h to be formulas and at least one of them not to be T. In $\mathrm{DC}_{\forall}$, Y is a fresh variable, and in $\mathrm{DC}_{\exists}$, sk is a fresh symbol and $\overline{\alpha}$ and $\overline{Y}_n$ consists of all the type and term variables occurring freely in $\exists x.\, g$. The rules described above are as given by Ganzinger and Stuber (adapted to our setting), with the omission of rules for negative literals (of the form $f \not\approx \mathsf{T}$), which are easy to derive and which can be found in their work [69].

Naive application of the DC rules can result in exponential blowup in problem size. To avoid this, we rename formulas f that have repeated occurrences by introducing predicates $\mathsf{p}\,\overline{X}_n$ replacing them, where $\overline{X}_n$ consists of all free variables of f. We keep the count of all nonatomic formulas occurring as either side of a literal. Before applying the DC rules on a clause $f \mathrel{\dot{\approx}} \mathsf{T} \vee C$, we check whether the number of occurrences of f exceeds the threshold k. If it does, based on the polarity of the literal $f \mathrel{\dot{\approx}} \mathsf{T}$, we add the clause $\mathsf{p}\,\overline{Y}_n \not\approx \mathsf{T} \vee f \approx \mathsf{T}$ (if the literal is positive) or $\mathsf{p}\,\overline{Y}_n \approx \mathsf{T} \vee f \not\approx \mathsf{T}$ (if the literal is negative), where $\overline{Y}_n$ are all free variables of f and p is a fresh symbol. Then, we replace the clause $f \mathrel{\dot{\approx}} \mathsf{T} \vee C$ by $\mathsf{p}\,\overline{Y}_n \mathrel{\dot{\approx}} \mathsf{T} \vee C$.

Before the number of occurrences of f is checked, we first check (using a fast, incomplete matching algorithm, used for simplification and subsumption) if there is a formula g, for which definition was already introduced, such that $\sigma(g) = f$, for some substitution σ. This check can have three outcomes. First, if the definition $\mathsf{q}\,\overline{X}_n$ was already introduced for g with the same polarity as $f \mathrel{\dot{\approx}} \mathsf{T}$, then f is replaced by $\sigma(\mathsf{q}\,\overline{X}_n)$. Second, if the definition was introduced, but with different polarity, we create the clause defining g with the missing polarity, and replace f with $\sigma(\mathsf{q}\,\overline{X}_n)$. Last, if there is no renamed formula g generalizing f, we check if the number of occurrences of f exceeds the threshold k.

In addition to reusing names for formula definitions, we reuse the Skolem symbols introduced by the $\mathrm{DC}_{\exists}$ rule. When $\mathrm{DC}_{\exists}$ is applied to $f = \exists x.\, f'$, we check if there is a Skolem $\mathsf{sk}\langle \overline{\alpha}_m \rangle\, \overline{Y}_n$ introduced for a formula $g = \exists x.\, g'$, such that $\sigma(g) = f$. If so, the symbol sk is reused and $\exists x.\, f'$ is replaced by $\{x \mapsto \sigma(\mathsf{sk}\langle \overline{\alpha}_m \rangle\, \overline{Y}_n)\}(f')$. Renaming and name reusing techniques are inspired by the VCNF algorithm described by Reger et al. [137].

Rules Cases and CasesSimp deal with Boolean terms, but we need to rely on extensionality reasoning to deal with λ-abstractions whose bodies have type o. Using the observation that the formula $\forall \overline{x}_n.\, f$ implies that $\lambda \overline{x}_n.\, f$ is functional-extensionally equal to $\lambda \overline{x}_n.\, \mathsf{T}$ (and similarly, if $\forall \overline{x}_n. \neg f$, then $\lambda \overline{x}_n.\, f \approx \lambda \overline{x}_n.\, \bot$), we designed the following rule (where $\overline{x}_n$ consists of all loose bound variables of f):

$$\frac{C[\lambda \overline{x}_n.\, f]}{(\forall \overline{x}_n.\, f) \not\approx \mathsf{T} \vee C[\lambda \overline{x}_n.\, \mathsf{T}] \qquad (\forall \overline{x}_n. \neg f) \not\approx \mathsf{T} \vee C[\lambda \overline{x}_n.\, \bot]}\,\textsc{Interpret}\lambda$$

5.4 Alternative Approaches

An alternative to modifications of the prover needed to support the rules described above is to treat Booleans as yet another theory. Since the theory of Booleans is finitely axiomatizable, stating those axioms instead of creating special rules might seem appealing. Another approach is to preprocess nested Booleans by hoisting them to the top level.

Axiomatization A simple axiomatization of the theory of Booleans is given by Bentkamp et al. [18]. Following their approach, we introduce the proxy type *bool*, which corresponds to o, to the signature. We define proxy symbols $\mathsf{t}, \mathsf{f}, \mathsf{not}, \mathsf{and}, \mathsf{or}, \mathsf{impl}, \mathsf{equiv}$, $\mathsf{forall}, \mathsf{exists}, \mathsf{choice}$, and eq which correspond to the homologous logical constants from Sect. 5.2. In their type declarations, o is replaced by *bool*.

To make this chapter self-contained we include the axioms from Bentkamp et al. [18]. Definitions of symbols are computational in nature: Symbols are characterized by their behavior on t and f. This also reduces interferences between different axioms. The axioms are listed as follows:

$$\mathsf{t} \not\approx \mathsf{f}$$
$$X \approx \mathsf{t} \vee X \approx \mathsf{f}$$
$$\mathsf{not}\,\mathsf{t} \approx \mathsf{f}$$
$$\mathsf{not}\,\mathsf{f} \approx \mathsf{t}$$
$$\mathsf{and}\,\mathsf{t}\,X \approx X$$
$$\mathsf{and}\,\mathsf{f}\,X \approx \mathsf{f}$$
$$\mathsf{or}\,\mathsf{t}\,X \approx \mathsf{t}$$
$$\mathsf{or}\,\mathsf{f}\,X \approx X$$
$$\mathsf{impl}\,\mathsf{t}\,X \approx X$$
$$\mathsf{impl}\,\mathsf{f}\,X \approx \mathsf{t}$$
$$X \not\approx Y \vee \mathsf{eq}\langle\alpha\rangle\,X\,Y \approx \mathsf{t}$$
$$X \approx Y \vee \mathsf{eq}\langle\alpha\rangle\,X\,Y \approx \mathsf{f}$$
$$\mathsf{equiv}\,X\,Y \approx \mathsf{and}\,(\mathsf{impl}\,X\,Y)\,(\mathsf{impl}\,Y\,X)$$
$$\mathsf{forall}\langle\alpha\rangle\,(\lambda x.\,\mathsf{t}) \approx \mathsf{t}$$
$$Y \approx (\lambda x.\,\mathsf{t}) \vee \mathsf{forall}\langle\alpha\rangle\,Y \approx \mathsf{f}$$
$$\mathsf{exists}\langle\alpha\rangle\,Y \approx \mathsf{not}\,(\mathsf{forall}\langle\alpha\rangle\,(\lambda x.\,\mathsf{not}\,(Y\,x)))$$
$$Y\,X \approx \mathsf{f} \vee Y\,(\mathsf{choice}\langle\alpha\rangle\,Y) \approx \mathsf{t}$$

Preprocessing Booleans Kotelnikov et al. extended VCNF, Vampire's algorithm for clausification, to support nested Booleans [98]. As explained in Sect. 3.8, we extended the clausification algorithm of Ehoh to support nested Booleans inspired by this VCNF extension. Zipperposition and Ehoh share the same clausification algorithm, enabling us to reuse the extension, with one notable difference: Unlike in Ehoh, not all nested Booleans different from variables, $\top$ and $\bot$ are removed. Namely, Booleans that are below λ-abstraction and contain λ-bound variables will not be preprocessed. They cannot be easily hoisted to the level of an atom in which they appear, since this process might leak any variables bound in the context in which the nested Boolean appears. Similar preprocessing techniques are used in other higher-order provers [174].

5.5 Examples

The TPTP library [157] contains thousands of higher-order benchmarks, many of them hand-crafted to point out subtle interferences of functional and Boolean properties of higher-order logic. In this section we discuss some problems from the TPTP library that illustrate the advantages and disadvantages of our approach.

During most of the 2010s, the core calculus of the best performing higher-order prover at CASC was tableaux—a striking contrast from the first-order part of the competition dominated by superposition-based provers. TPTP problem `SET557^1` might shed some light on why tableaux-based provers excel on higher-order problems. This problem conjectures

that there is no surjection from a set to its power set:

$$\neg(\mathbf{\exists} x.\mathbf{\forall} y.\mathbf{\exists} z.\, x\, z \approx y)$$

After negating the conjecture and clausification this problem becomes $\mathsf{sk}_1\,(\mathsf{sk}_2\,Y) \approx Y$ where sk_1 and sk_2 are Skolem symbols. Then, the ArgCong rule [18] which applies fresh variable W to both sides of the equation can be used, yielding clause $C = \mathsf{sk}_1\,(\mathsf{sk}_2\,Y)\,W \approx Y\,W$. Superposition-based higher-order theorem provers (such as Leo-III, Vampire, and Zipperposition) split this clause into two clauses $C_1 = \mathsf{sk}_1\,(\mathsf{sk}_2\,Y)\,W \not\approx \top \vee Y\,W \approx \top$ and $C_2 = \mathsf{sk}_1\,(\mathsf{sk}_2\,Y)\,W \approx \top \vee Y\,W \not\approx \top$. This clausification step makes the problem considerably harder. Namely, the clause C instantiated with the substitution $\{Y \mapsto \lambda x.\neg(\mathsf{sk}_1\,x\,x), W \mapsto \mathsf{sk}_2\,(\lambda x.\neg(\mathsf{sk}_1\,x\,x))\}$ yields the empty clause. However, if the original clause is split into two as described above, Zipperposition will rely on the PI rule to instantiate Y with an imitation of $\neg$ and on equality factoring to further instantiate this approximation. These desired inferences need to be applied on both new clauses and represent only a fraction of inferences that can be done with C_1 and C_2, reducing the chance of a successful proof attempt. Rule BoolER imitates the behavior of a tableaux prover: It essentially rewrites the clause C into $\neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,Y)\,W) \not\approx Y\,W$, which makes finding the necessary substitution easy and does not require a clausification step.

Combining the rule (Bool)ER with dynamic clausification is very fruitful, as the benchmark `SYO033^1` illustrates. This problem contains the single conjecture

$$\mathbf{\exists} x.\mathbf{\forall} y.\, x\, y \leftrightarrow (\mathbf{\forall} z.\, y\, z)$$

The problem is easily solved if the variable x is replaced with the constant $\mathsf{\forall}$. Moreover, the prover does not have to blindly guess this instantiation. Instead, pretending that bound variables are free, it can obtain it by unifying $X\,Y$ with $\mathbf{\forall}\,Y$ (which is the η-short form of $\mathbf{\forall} z.\, Y\, z$). However, when the problem is clausified, all quantifiers are removed. Then, Zipperposition finds the proof only if an appropriate instantiation mode of PI is used, and if both clauses resulting from clausifying the negated conjecture are appropriately instantiated. In contrast, dynamic clausification derives the clause $X\,(\mathsf{sk}\,X) \not\approx \mathsf{\forall}\,(\mathsf{sk}\,X)$ from the negated conjecture in three steps. Then, equality resolution results in an empty clause, swiftly finishing the proof without any explosive inferences. This effect is even more pronounced on problems `SYO287^5` and `SYO288^5`, in which a critical proof step consists of instantiating a variable with imitations of $\mathsf{\vee}$ and $\mathsf{\wedge}$. In configurations that do not use dynamic clausification and BoolER, Zipperposition times out in any reasonable time limit; with those two options it solves these two problems in less than 100 ms.

In some cases, it is better to preprocess the problem. For example, TPTP problem `SYO500^1.005` contains many nested Boolean terms:

$$\mathsf{f}_0\,(\mathsf{f}_1\,(\mathsf{f}_1\,(\mathsf{f}_1\,(\mathsf{f}_2\,(\mathsf{f}_3\,(\mathsf{f}_3\,(\mathsf{f}_3\,(\mathsf{f}_4\,\mathsf{a})))))))) \approx \mathsf{f}_0\,(\mathsf{f}_0\,(\mathsf{f}_0\,(\mathsf{f}_1\,(\mathsf{f}_2\,(\mathsf{f}_2\,(\mathsf{f}_3\,(\mathsf{f}_4\,(\mathsf{f}_4\,(\mathsf{f}_4\,\mathsf{a})))))))))$$

In this problem, all functions f_i are of type $o \to o$, and constant a is of type o. FOOL unfolding of nested Boolean terms will result in an exponential blowup in the problem size. However, superposition-based theorem provers are well equipped for this issue: Their clausification algorithms use smart simplifications and formula renaming to mitigate these effects. Moreover, when the problem is preprocessed, the prover is aware of the problem

	a	lo	li
b	*1646*	**1648**	1640
b_c	1644	1645	1644

Figure 5.1: Effect of the Cases(Simp) rule on success rate

size before the proving process starts and can adjust its heuristics properly. E, Zipperposition, and Vampire, instructed to perform FOOL unfolding, solve the problem swiftly, using their default modes. However, if the problem is not preprocessed, Zipperposition struggles to prove it using Cases(Simp), and due to the large number of (redundant) clauses it creates, succeeds only if specific heuristic choices are made.

5.6 Evaluation

We performed an extensive evaluation to determine the usefulness of our approach. As our benchmark set, we used all 2606 monomorphic theorems from the TPTP 7.2.0 library, given in THF format. All of the experiments described in this section were performed on StarExec [154] servers with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz. The evaluation is separated in two parts that answer different questions: How useful are the new rules? How does our approach compare with state-of-the-art higher-order provers?

Evaluation of the Rules For this part of the evaluation, we fixed a single well-performing Zipperposition configuration called *base* (b). Since we are testing a single configuration, we used the CPU time limit of 15 s—roughly the time a single configuration is given in Zipperposition's portfolio mode when participating in CASC. Configuration b uses the pragmatic variant pv^2_{1121} (Sect. 4.8) of the unification algorithm described in Chapter 4. It enables the BoolSimp rule, the EC rule, the PI rule in *Pragmatic* mode with $k = 2$, rules ElimLeibniz and ElimPredVar, the BoolER rule, and the BoolEF rules. To evaluate the usefulness of all rules described above, we enabled, disabled, or changed the parameters of a single rule, while keeping all other parameters of b intact. In figures that contain sufficiently many different configurations, cells are of the form $n(m)$ where n is the total number of proved problems by a particular configuration and m is the number of unique problems that a given configuration solved, compared to the other configurations in the same figure. Intersections of rows and columns denote the corresponding combination of parameters. The result for the base configuration is written in *italics*; the best result is written in **bold**.

First, we tested different parameters of the Cases and CasesSimp rules. In Figure 5.1 we report the results. The columns correspond to three possible options to choose the subterm on which the inference is performed: a stands for any eligible subterm, lo and li stand for leftmost outermost and leftmost innermost subterms, respectively. The rows correspond to two different rules: b is the base configuration, which uses CasesSimp, and b_c swaps this rule for Cases. Although the margin is slim, the results show it is usually preferable to select the leftmost-outermost subterm.

	–PI	b_p	b_f	$b_\wedge$	$b_\vee$	$b_\approx$	$b_\neg$	$b_{\forall\exists}$
$k = 1$		**1648**	1628	1637	1634	1630	1641	1637
$k = 2$	1636	*1646*	1629	1636	1631	1627	1638	1634
$k = 8$		1643	1625	1633	1631	1623	1637	1635

Figure 5.2: Effect of the PI rule on success rate

	–EL	+EL
–EPV	1584 (0)	1644 (0)
+EPV	1612 (0)	***1646 (0)***

Figure 5.3: Effect of Leibniz equality elimination rules

	–BEF	+BEF
–BER	1644 (2)	1643 (0)
+BER	1645 (0)	***1646 (0)***

Figure 5.4: Effect of BoolER and BoolEF rules

Second, we evaluated all the modes of the PI rule with three values for parameter k: 1, 2, and 8 (Figure 5.2). The columns denote, from left to right: disabling the PI rule, *Pragmatic* mode, *Full* mode, and $Imit_\star$ modes with appropriate logical symbols. The rows denote different values of k. The results show that different values for k have a modest impact on the success rate. The raw data reveal that when we focus our attention to configurations with $k = 2$, mode *Full* can solve ten problems no other mode (including disabling the PI rule) can. Modes $Imit_\wedge$ and *Pragmatic* solve two problems whereas $Imit_\vee$ solves one problem uniquely. This result suggests that, even though this is not evident from Figure 5.2, the sets of problems solved by different modes differ somewhat.

Figure 5.3 gives results of evaluating rules that treat Leibniz equality on the calculus level: EL stands for ElimLeibniz, whereas EPV denotes ElimPredVar; signs – and + denote that the corresponding rule is removed from or added to configuration b, respectively. Disabling both rules severely lowers the success rate. The results suggest that including ElimLeibniz is beneficial to performance.

Similarly, Figure 5.4 shows the merits of excluding (–) or including (+) BoolER (BER) and BoolEF (BEF) rules. Our expectations were that inclusion of those two rules would have a significant impact on the success rate. It turns out that, in practice, most of the effects of these rules could be achieved using a combination of the PI rule and the rules of the superposition calculus.

Combining BER and BEF rules with dynamic clausification is more useful: When rule IC is replaced by rule DC, the success rate increases to 1660 problems, compared to 1646 problems solved by b. We also discovered that reasoning with choice is useful: When rule Choice is enabled, the success rate increases to 1653. We determined that including or excluding the conclusion D of Choice, after it is simplified, makes no difference. Counterintuitively, disabling the BoolSimp rule results in 1640 problems, which is only 6 problems short of configuration b. Disabling the Abs and Interpretλ rules results in solving 25 and 31 problems fewer, respectively. The raw data show that in total, using configurations from Figure 5.1 to Figure 5.4, 1682 problems can be solved.

Last, we compared our approach to alternatives. Axiomatizing Booleans brings Zipperposition down to a grinding halt: only 1106 problems can be solved using this mode.

	CVC4	Leo-III	Satallax	Vampire	Zipperposition
uncoop	1806 (5)	1627 (0)	2067 (0)	1924 (7)	1980 (0)
coop	–	2085 (3)	**2214 (9)**	–	2190 (17)

Figure 5.5: Comparison with other higher-order provers

On the other hand, preprocessing is fairly competitive: it solved only 8 problems fewer than the b configuration.

Comparison with Other Higher-Order Provers We compared Zipperposition with all higher-order theorem provers that took part in higher-order division of the 2019 edition of CASC [160]: CVC4 1.8 prerelease [12], Leo-III 1.4 [153], Satallax 3.4 [39], and Vampire-THF 4.4 [100]. In this part of the evaluation, Zipperposition used the portfolio mode that runs configurations in different time slices. We set the CPU time limit to 180 s, the time allotted to each prover at the 2019 edition of CASC.

Leo-III and Satallax are cooperative theorem provers—they periodically invoke first-order provers to finish the proof attempt. Leo-III uses CVC4, E, and iProver [97] as backends, and Satallax uses Ehoh as backend. Zipperposition can use Ehoh as backend as well. To test how successful each calculus is, we ran the cooperative provers in two versions: *uncoop*, which disables backends, and *coop*, which uses all supported backends.

In both uncooperative and cooperative mode, Satallax is the winner. Zipperposition comes in close second, showing that our approach is a promising basis for further extensions. Indeed, as is shown in Chapter 6, with fine-tuning of the heuristics and further extensions to the calculus, this approach can outperform all other competitive provers with a large margin.

5.7 Discussion

Our work is primarily motivated by the goal of closing the gap between higher-order "hammer" or software verifier frontends and first-order backends. A considerable amount of research effort has gone into making the translations of higher-order logic as efficient as possible. Descriptions of hammers like HOLyHammer [89] and Sledgehammer [131] for Isabelle contain details of these translations. Software verifiers Boogie [102] and Why3 [33] use similar translations.

Established higher-order provers like Leo-III and Satallax have been optimized to perform well on TPTP; however, recent evaluations (including the one in Sect. 3.9) show that on Sledgehammer problems they are outperformed by translations to first-order logic [12, 18]. Those two provers are built from the ground up as higher-order provers—treatment of exclusively higher-order issues such as extensionality or choice is built into them often using explosive rules. Those explosive rules might contribute to their suboptimal performance on mostly first-order Sledgehammer problems.

In contrast, the approach taken in this thesis is to start with a first-order prover and gradually extend it with higher-order features. The work performed in the context of the Matryoshka project [28], in which I participated, resulted in adding support for λ-free

higher-order logic with Booleans to E (Chapter 3) and veriT [12], and adding support for Boolean-free higher-order logic to Zipperposition. Many authors of state-of-the-art first-order provers have implemented some form of support for higher-order reasoning. This is true both for SMT solvers, witnessed by the recent extension of CVC4 and veriT [12], and for superposition provers, witnessed by the extension of Vampire [24]. All of those approaches were arguably more focused on functional aspects of higher-order logic, such as λ-binders and function extensionality, than on Boolean aspects such as Boolean subterms and Boolean extensionality. A notable exception is work by Kotelnikov et al. that introduced support for Boolean subterms to first-order Vampire [98, 99].

The main merit of our approach is that it combines two successful complementary approaches, λSup and FOOL paramodulation, to support features of higher-order logic that have not been combined before in a modular way. It is based on λSup, a calculus that generalizes the highly successful first-order superposition calculus. It incurs around 1% of overhead on first-order problems compared with classic superposition [18].

5.8 Conclusion

We presented a pragmatic approach to support Booleans in a modern automatic prover for clausal higher-order logic. Our approach combines previous research efforts that extended first-order provers with complementary features of higher-order logic. It also proposes some solutions for issues that emerge with this combination. The implementation shows a clear improvement over previous techniques and a competitive performance.

What our work misses is an overview of heuristics that can be used to curb the explosion incurred by some of the rules described in this chapter. In the next chapter, we explore exactly this topic.

6

Making Higher-Order Superposition Work

Joint work with Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret

Superposition is among the most successful calculi for first-order logic. Its extension to higher-order logic introduces new challenges such as infinitely branching inference rules, new possibilities such as reasoning about Booleans, and the need to curb the explosion of specific higher-order rules. We describe techniques that address these issues and extensively evaluate their implementation in the Zipperposition theorem prover. Largely thanks to their use, Zipperposition won the THF division of the CASC competition in 2020 and 2021.

In this work I was the main designer of all presented techniques, with the exception of inference streams that were designed by Alexander Bentkamp and Sophie Tourret. Alexander Bentkamp and Jasmin Blanchette also discussed many of the techniques with me and suggested important updates. Visa Nummelin worked on the implementation of FOOL preprocessing. Simon Cruanes is the original developer of Zipperposition and provided us with invaluable knowledge.

6.1 Introduction

The landscape of higher-order proving techniques based on the extension of efficient first-order ones has expanded tremendously in the late 2010s and early 2020s. As mentioned in Sect. 2.5.6, we have implemented three higher-order calculi—λfSup, λSup, and *o*λSup—that extend first-order superposition in a graceful way. Bhayat and Reger also gracefully extended superposition to higher-order logic using SKBCI combinators [25], resulting in a calculus called combinatory superposition. Significant progress has been made on the SMT front as well [11].

In 2019 we tested for the first time if the idea of gracefully extending first-order provers to higher-order logic really improves the state of the art. We implemented λSup [18] in Zipperposition 1.5 with basic heuristics and rudimentary extensions of the calculus to deal with Booleans. It finished third at that year's THF division of CASC competition [160], 12 percentage points behind the winner, the tableau prover Satallax 3.4 [39].

Studying the competition results, we found that higher-order tableaux have some advantages over higher-order superposition. To bridge the gap, we developed techniques and heuristics that simulate tableaux in the context of saturation. We implemented them in Zipperposition 2, which took part at the higher-order division of CASC [161] in 2020. This time, our prover won the division, proving 84% of the problems, a whole 20 percentage points ahead of the runner-up, Satallax 3.4.

In this chapter, we describe the main techniques that explain this reversal of fortunes. They cover most parts of a modern higher-order theorem prover, from preprocessing to additional calculus rules to heuristics to backend integration. Compared to the previous chapter, in which we discussed rules used to treat Boolean terms, in this chapter we use a newer version of Zipperposition, based on a newer calculus. Instead of λSup augmented with ad hoc Boolean rules, we work with *o*λSup [17], a principled extension of superposition to full higher-order logic, including an interpreted Boolean type.

Many higher-order problems use symbol definitions extensively to decrease the verbosity of representation. We describe several ways to exploit the definitions, such as turning them into rewrite rules (Sect. 6.3). By working on formulas rather than clauses, tableau techniques take a more holistic view of a higher-order problem. Through its support for delayed clausification and, more generally, calculus-level formula manipulation, *o*λSup enables us to simulate most successful tableau techniques in a saturating prover (Sect. 6.4). This calculus also supports *Boolean selection functions*, a mechanism that allows us to choose which Boolean subterms to perform inferences on first. We implemented some Boolean selection functions and evaluated them (Sect. 6.5).

The main implementation challenge of both λ-superposition variants compared with combinatory superposition is that they rely on rules that enumerate possibly infinite sets of unifiers. We describe a mechanism that interleaves infinitely branching inferences with the standard saturation process (Sect. 6.6). The prover retains the same behavior as before on first-order problems, smoothly scaling with increasing numbers of higher-order clauses. We also propose heuristics to curb the explosion induced by highly prolific calculus rules (Sect. 6.7).

Using first-order backends to finish the proof is common practice in higher-order reasoning. Since *o*λSup coincides with standard superposition on first-order clauses, invoking backends may seem redundant; yet Zipperposition is nowhere as efficient as E [147] or

Vampire [100], so invoking a more efficient backend does make sense. We describe how to achieve a balance between allowing native higher-order reasoning and delegating reasoning to a backend (Sect. 6.8). Finally, we compare Zipperposition 2 with other provers on all monomorphic higher-order TPTP benchmarks [157] to perform a more extensive evaluation than at CASC (Sect. 6.9). Our evaluation corroborates the competition results.

6.2 Background and Setting

We focus on monomorphic higher-order logic, defined in Sect. 2.3. However, the techniques can easily be extended with rank-1 polymorphism [88]. Indeed, Zipperposition already supports some of them polymorphically. Further, we use exactly the same notation for this logic and superposition calculus as introduced in Chapter 2. Since we are working with extensions of superposition, we assume a clausal structure (Sect. 2.4). As in the previous chapter, literals of clauses can contain arbitrary higher-order terms, including formulas. At CASC, most theorem provers, including Zipperposition, are invoked using a sequence of different configurations (possibly in parallel) until either the time limit is reached or a proof is found. This sequence is usually called a *portfolio*.

Higher-Order Calculi We briefly introduced the $o\lambda$Sup calculus [17] in Sect. 2.5.6. It is a refutationally complete inference system and redundancy criterion for higher-order logic with rank-1 polymorphism, Hilbert choice, and functional and Boolean extensionality. Unlike λSup, this calculus does not require axioms defining the logical symbols to cope with formulas. Instead, it includes Boolean inference rules that mimic superposition from such axioms into Boolean subterms, while avoiding the explosion incurred by adding these axioms to the proof state. It also includes rules that simulate Boolean inferences below applied variables. Both sets of rules are disabled or replaced with incomplete, ad hoc rules described in the previous chapter in most configurations of the CASC portfolio. A new feature of the calculus that we explore in detail is the ability to select Boolean subterms to restrict Boolean and superposition inferences.

In contrast to both λ-superposition variants, combinatory superposition does not require enumerating elements of CSU to compute results of inferences. Instead, it avoids computing CSUs by using a form of first-order unification. Essentially, it enumerates higher-order terms using rules that instantiate applied variables with partially applied combinators from the complete combinator set $\{\mathsf{S}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{I}\}$. This calculus is the basis of Vampire 4.5 [25], which finished closely behind Satallax 3.4 at the higher-order division of CASC in 2020.

A different, very successful calculus is Satallax's SAT-guided tableaux [9]. Satallax was the leading higher-order prover of the 2010s. Its simple and elegant tableaux avoid deep superposition-style rewriting inferences. Nevertheless, our working hypothesis for the past years has been that superposition would likely provide a stronger basis for higher-order reasoning. Other competing higher-order calculi include SMT (implemented in CVC4 [11, 12]) and extensional paramodulation (implemented in Leo-III [153]).

Experimental Setup To assess our techniques, we carried out experiments with Zipperposition 2. We used two sets of benchmarks: all 2851 monomorphic higher-order prob-

lems from the TPTP library [157] version 7.4.0 (labeled *TPTP* in the figures of this chapter) and 1253 Sledgehammer-generated monomorphic higher-order problems (labeled *SH*). Although some techniques support polymorphism, we uniformly used monomorphic benchmarks.

We fixed a *base* configuration of Zipperposition parameters as a baseline for all comparisons. This is an incomplete, pragmatic configuration of *o*λSup using heuristics expected to perform well on a wide range of problems. The set of parameters used for the baseline configuration in this chapter differs from the one in the previous chapter. In each experiment, we varied the parameters associated with a specific technique to evaluate it. The experiments were run on StarExec Miami [154] servers, equipped with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz. Unless otherwise stated, we used a CPU time limit of 15 s, roughly the time each configuration is given in the CASC portfolio mode. The raw evaluation results are available online.[1]

6.3 Preprocessing Higher-Order Problems

The TPTP library contains thousands of higher-order problems. Despite their diversity, they have a markedly different flavor from the TPTP first-order problems. Notably, they extensively use the `definition` role to identify universally quantified equations (and equivalences) that define symbols. Definitions $s \approx t$ (or $(s \leftrightarrow t) \approx \mathsf{T}$) can be replaced by rewrite rules $s \longrightarrow t$, using the orientation given in the input problem. If there are multiple definitions for the same symbol, only the first one is replaced by a rewrite rule. Then, whenever a clause is picked in the given clause procedure, it is rewritten using the collected rules. Alternatively, we can rewrite the input formulas as a preprocessing step. This ensures that the input clauses will be fully simplified when the proving process starts and no defined symbols will occur in clauses, which usually helps the heuristics.

Since the TPTP format enforces no constraints on definitions, rewriting might diverge. To ensure termination, we limit the number of applied rewrite steps. In practice, most TPTP problems are well behaved: Only one definition is given for each symbol, and the definitions are acyclic.

Turning the defining equations into rewrite rules, unfolding the definitions, and β-reducing the result can eliminate all of a problem's higher-order features, making it susceptible to first-order methods. However, this can inflate the problem beyond recognition and compromise the refutational completeness of superposition.

Example 6.1. Removing higher-order features of a problem can have adverse effects. Consider the TPTP problem `NUM636^3`, which defines the predicate m as $\lambda x.\, \mathsf{s}\, x \not\approx x$ and states its conjecture as $\forall x.\, \mathsf{m}\, x$, where s is the standard Peano-style natural number successor constructor. When this definition is kept as is, the prover can superpose from either m or its definition into the (clausified) induction axiom, which is also given in the problem, and quickly prove the conjecture, without using any advanced inductive reasoning. In contrast, when the definition is unfolded and the problem is β-reduced, both m and the corresponding λ-abstraction disappear, forcing the prover to guess the correct instantiation for the induction axiom.

[1] `http://doi.org/10.5281/zenodo.5007440`

We describe two techniques to mitigate these issues. The first one is based on the observation that in practice, the explosion associated with definition unfolding mostly manifests itself on definitions of nonpredicate symbols. In some cases, it is preferable to rely on superposition's term order and the powerful simplification engine to rewrite the proof state rather than to blindly rewrite definitions. On the other hand, superposition's reasoning with equivalences is often inadequate [17, 69]. Thus, it makes sense to treat only predicate definitions as rewrite rules.

The second technique aims at preserving completeness: We can try to choose the term order that parameterizes superposition, as one that orients as many definitions as possible, and rely on demodulation to simplify the proof state. Usually, an instance of the Knuth–Bendix order (KBO) [95] is used. KBO compares terms by first comparing their weights, which is the sum of all the weights assigned to the symbols it contains. Given a symbol weight assignment $\mathscr{W}$, we can update it so that it orients acyclic definitions from left to right assuming that they are of the form $\mathsf{f}\,\overline{X}_m \approx \lambda \overline{Y}_n.\, t$, where the only free variables in t are $\overline{X}_m$, no free variable is repeated or appears applied in t, and f does not occur in t. Then we traverse the symbols f that are defined by such equations following the dependency relation, starting with a symbol f that does not depend on any other defined symbol. For each f, we set $\mathscr{W}(\mathsf{f})$ to $w + 1$, where w is the maximum weight of the right-hand sides of f's definitions, computed using $\mathscr{W}$. By construction, for each equation the left-hand side is heavier. Thus, the equations are orientable from left to right.

Example 6.2. Many of the problems in the TPTP library's `LCL` category encode modal logic in higher-order logic. More complex modal operators (such as implication and equivalence) are defined in terms of basic connectives (such as negation and disjunction). Some of the definitions present in the problems are $\mathsf{mnot} := \lambda p\, x.\, \neg\, p\, x$, $\mathsf{mor} := \lambda p\, q\, x.\, p\, x \vee q\, x$, and $\mathsf{mimplies} := \lambda p\, q.\, \mathsf{mor}\,(\mathsf{mnot}\, p)\, q$. Assuming that the weight of λ, bound variables, and basic connectives is 2, we can orient equations using the approach above described as follows. Starting from symbols that do not depend on the other ones, we set $\mathscr{W}(\mathsf{mnot}) = 11$ and $\mathscr{W}(\mathsf{mor}) = 17$. Then, we use these values to set $\mathscr{W}(\mathsf{mimplies}) = 37$. Clearly, these weights enable us to orient all definitions from left to right.

Evaluation and Discussion We designed and evaluated the following strategies for handling `definition` axioms:

pre-RW	rewrite all definitions as a preprocessing step;
in-RW	rewrite all definitions during the saturation, as an inprocessing step;
o-RW	rewrite only predicate definitions, during preprocessing;
o-RW+KBO	like *o*-RW but with adjusted KBO weights for the remaining definitions;
no-RW	no special treatment of definitions;
no-RW+KBO	like *no*-RW but adjusting KBO weights for all definitions.

The results are given in Figure 6.1. In all the figures in this chapter, each cell gives the number of proved problems, and cells marked with ⋆ correspond to the base configuration.

	pre-RW	in-RW	*o*-RW	*o*-RW+KBO	no-RW	no-RW+KBO
TPTP	**1635***	1619	1620	1621	1298	1296

Figure 6.1: Impact of the definition rewriting method

The highest number in a category is typeset in bold. SH benchmarks are not included because they do not contain the `definition` role.

The four configurations in which definitions are treated as rewrite rules perform much better than the other two. In contrast, adjusting KBO weights gives no substantial improvement: Looking at raw data, we found only two problems proved by *o*-RW+KBO but not by *o*-RW in which the feature was used in the proof. For no-RW and no-RW+KBO, the two-problem difference may just be noise. Even though it proves fewer problems, the configuration *o*-RW has some advantages over pre-RW: It proves 16 problems that pre-RW does not, three of which have a TPTP difficulty rating of 1. Difficulty rating is a number from 0 to 1, proportional to the number of state-of-the-art provers that attempted, but failed to solve to problem [162].

Rewriting after clausification avoids getting stuck in rewriting parts of the proof state that might not contribute to the proof. In practice, we noticed that rewriting can be so expensive that the prover can spend all allotted CPU time in the preprocessing phase. The evaluation results confirm this observation: There are 64 problems proved by in-RW but not by pre-RW. Moreover, there are 41 problems that can be proved only by in-RW but not by any other above described configuration.

6.4 Reasoning about Formulas

Higher-order logic identifies formulas with terms of Boolean type. To prove a conjecture, we often need to instantiate a variable with the right predicate. Finding this predicate can be easier if the problem is not clausified. Consider the conjecture $\exists f.\, f\,\mathsf{p}\,\mathsf{q} \leftrightarrow \mathsf{p} \wedge \mathsf{q}$. Expressed in this form, the formula is easy to prove by taking $f := \lambda x\, y.\, x \wedge y$. By contrast, guessing the right instantiation for the negated, clausified form $\neg F\,\mathsf{p}\,\mathsf{q} \vee \neg\mathsf{p} \vee \neg\mathsf{q}, F\,\mathsf{p}\,\mathsf{q} \approx \mathsf{T} \vee \mathsf{p} \approx \mathsf{T}, F\,\mathsf{p}\,\mathsf{q} \approx \mathsf{T} \vee \mathsf{q} \approx \mathsf{T}$ is more challenging. One of the strengths of higher-order tableau provers is that they do not clausify the input problem. This might partly explain Satallax's dominance in the THF division of CASC competitions until the 2020 edition of CASC.

The *o*λSup calculus supports *delayed clausification rules* that insert the formulas of a problem into the proof state in their original, nonclausified form, and clausify them gradually. Delayed clausification allows the prover to analyze the syntactic structure of formulas during saturation, whereas the more traditional approach of *immediate clausification* applies a standard clausification algorithm [126] both as a preprocessing step and whenever predicate variables are instantiated.

An earlier evaluation of the *o*λSup calculus [17] showed that the *outer* variant of delayed clausification substantially increases this calculus's performance. The outer variant clausifies top-level logical symbols, proceeding from the outside inwards; this method corresponds to the one described in the previous chapter. For example, a clause $C \vee \neg(\mathsf{p} \wedge \mathsf{q})$ is transformed into $C \vee \neg\mathsf{p} \vee \neg\mathsf{q}$. The calculus also supports *inner* delayed clausification,

which does not enforce clausifying logical symbols in top-down direction, and uses only the generating calculus rules to clausify problems. Even though this is the laziest approach to clausification, the earlier evaluation [17] showed that this approach is inefficient. Thus, we focus only on the outer rules.

Delayed clausification rules can be used as inference rules (which add conclusions to the passive set) or as simplification rules (which delete premises and add conclusions to the passive set). Inferences are more flexible, as all intermediate clausification states will be stored in the proof state, at the cost of producing many clauses. Simplifications produce fewer clauses, but risk destroying informative syntactic structure. Since clausifying equivalences can destroy a lot of syntactic structure [69], we never apply simplifying rules on them.

Delayed clausification can interfere with clause splitting techniques. Zipperposition supports a lightweight variant of AVATAR [167], an architecture that partitions the search space by splitting clauses into variable-disjoint subclauses. This lightweight AVATAR is described by Ebner et al. [58, Sect. 7]. Combining it with delayed clausification makes it possible to split a clause $(\varphi_1 \vee \cdots \vee \varphi_n) \approx \mathsf{T}$, where the φ_i's are arbitrarily complex formulas that share no free variables with each other, into clauses $\varphi_i \approx \mathsf{T}$. To finish the proof, it suffices to derive the empty clause under each assumption $\varphi_i \approx \mathsf{T}$. Since the split is performed at the formula level, this technique resembles tableaux, but it exploits the strengths of superposition, such as its powerful redundancy criterion and simplification machinery, to close the branches.

Beyond splitting, interleaving clausification and saturation allows us to simulate another tableau-inspired technique. Whenever dynamic clausification substitutes a fresh variable X for a predicate variable x in a clause of the form $(\forall x.\, \varphi) \approx \mathsf{T} \vee C$, yielding $\{x \mapsto X\}(\varphi) \approx \mathsf{T} \vee C$, we can create additional clauses in which x is replaced with $t \in \mathit{Inst}$, where *Inst* is a set of heuristically chosen terms. This set contains λ-abstractions whose bodies are formulas and that occur in activated clauses; it also contains *primitive instantiations*—that is, imitations (in the sense of higher-order unification) of logical symbols that approximate the shape of a predicate that can instantiate a predicate variable. Primitive instantiations are described in Sect. 5.3.

Since a new term t can be added to *Inst* after a clause with a quantified variable of t's type has been activated, we remember the clauses $\{x \mapsto X\}(\varphi) \approx \mathsf{T} \vee C$ and instantiate them when *Inst* is extended. Conveniently, these instantiated clauses are not recognized as subsumed by Zipperposition, which uses an optimized, incomplete higher-order subsumption algorithm.

Given a disequation $\mathsf{f}\,\overline{s}_n \not\approx \mathsf{f}\,\overline{t}_n$, the *abstraction* of s_i is $\lambda x.\, u \not\approx v$, where u is obtained by replacing all occurrences of s_i in $\mathsf{f}\,\overline{s}_n$ with x and v is obtained by replacing all occurrences of s_i in $\mathsf{f}\,\overline{t}_n$ with x. For an equation $\mathsf{f}\,\overline{s}_n \approx \mathsf{f}\,\overline{t}_n$, the analogous abstraction is $\lambda x.\, \neg(u \not\approx v)$. Adding abstractions of the literals occurring in the conjecture to *Inst* provides useful instantiations for formulas such as induction principles of datatypes. As the conjecture is negated in refutational theorem proving, the equation's polarity is inverted in the abstraction.

Example 6.3. The clausified conjecture of the problem `DAT056^2` [156] from the TPTP library is $\mathsf{ap}\,\mathsf{xs}\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,\mathsf{xs}\,\mathsf{ys})\,\mathsf{zs}$, where ap is the list append operator defined recursively on its first argument and xs, ys, and zs are of list type. Abstracting xs from the disequation yields $t = \lambda \mathit{xs}.\, \mathsf{ap}\,\mathit{xs}\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,\mathit{xs}\,\mathsf{ys})\,\mathsf{zs}$, which is added to *Inst*. In-

		+LA	–LA
TPTP	IC	1616	1635*
	DCI	1507	1532
	DCS	1668	**1703**
SH	IC	425	452*
	DCI	362	385
	DCS	441	**457**

Figure 6.2: Impact of clausification and lightweight AVATAR

cluded in the problem is the induction axiom for the list datatype: $\forall p.\, p\,\mathsf{nil} \wedge (\forall x\, xs.\, p\, xs \longrightarrow p\,(\mathsf{cons}\, x\, xs)) \longrightarrow \forall xs.\, p\, xs$, where nil and cons have the usual meanings. Instantiating p with t and using the ap definition, we can prove $\forall xs.\, \mathsf{ap}\, xs\,(\mathsf{ap}\, \mathsf{ys}\, \mathsf{zs}) \approx \mathsf{ap}\,(\mathsf{ap}\, xs\, \mathsf{ys})\, \mathsf{zs}$, from which we easily derive a contradiction.

Evaluation and Discussion The base configuration (*base*) uses immediate clausification (IC) and disables lightweight AVATAR (–LA). To test the merits of delayed clausification, we vary *base*'s parameters along two axes: We choose immediate clausification (IC), delayed clausification as inference (DCI), or delayed clausification as simplification (DCS), and we either enable (+LA) or disable (–LA) lightweight AVATAR. Neither of the configurations uses instantiation with terms from *Inst*.

Figure 6.2 shows that using delayed clausification as simplification greatly increases the success rate, regardless of whether lightweight AVATAR is used. Using delayed clausification as inference has the opposite effect on both problem sets, presumably due to the large number of clauses it creates. By manually inspecting the proofs found by the DCS configuration, we noticed that a main reason for its success is that it does not simplify away equivalences. Overall, lightweight AVATAR harms performance, but the sets of problems proved with and without it are vastly different. For example, the IC+LA configuration proves 38 problems not proved by IC–LA (i.e., *base*) on TPTP benchmarks and 14 such problems on SH benchmarks.

The Boolean instantiation technique presented above requires delayed clausification. We assessed it in the best configuration from Figure 6.2, DCS–LA. With this change (+BI), Zipperposition proves 1700 TPTP problems and 456 SH problems. On TPTP, even though +BI solves three problems fewer than DCS–LA, it is very useful: 41 problems can be proved with +BI but not with DCS–LA. Conversely, 44 problems are solved with DCS–LA, but not with +BI, which suggests that Boolean instantiation can be explosive. One of the problems Boolean instantiation helps solve is `NUM636^2` (a reencoding of `NUM636^3`, which is discussed in Example 6.1). It conjectures that $\forall x.\, \mathsf{s}\, x \not\approx x$, where x ranges over Peano-style numbers specified by z and s. The given axioms are the induction principle $\forall p.\, p\, \mathsf{z} \wedge \forall x.\, (p\, x \longrightarrow p(\mathsf{s}\, x)) \longrightarrow \forall x.\, p\, x$, injectivity $\forall xy.\, \mathsf{s}\, x \approx \mathsf{s}\, y \longrightarrow x \approx y$, and distinctness $\forall x.\, \mathsf{s}\, x \not\approx \mathsf{z}$. The conjecture is easily proved if Boolean instantiation is enabled: Even though the conjecture literal cannot be abstracted, instantiating p with the term $\lambda x.\, \mathsf{s}\, x \not\approx x$ used in the encoding of the (nonclausified) conjecture leads to a proof in just 22 given clause

loop iterations. Zipperposition also finds a proof using the DCI–LA configuration, but this requires 294 iterations.

The +BI configuration proves 18 TPTP problems no other configuration from Figure 6.2 can prove. Among these is `DAT056^2` (Example 6.3). In contrast, on SH benchmarks, only six problems are proved using +BI and not DCS–LA. For all these problems, Boolean instantiation does not appear in the proof, suggesting that this result is due to the randomness in the evaluation environment. The fact that BI has no effect on SH benchmarks is to be expected because Sledgehammer deliberately tries to exclude induction rules from the problem by excluding lemmas whose name contains the substring `.induct` and that contain predicate variables. Therefore, BI applies to fewer clauses.

6.5 Exploring Boolean Selection Functions

As discussed in Sect. 2.5, superposition calculi are parameterized by a literal selection function and a term order that help prune considerable swaths of the search space without jeopardizing completeness. The core inferences apply only to a clause's *eligible* literals, defined as either the clause's selected literals or, if none are selected, the clause's literals that are maximal with respect to the term order. To further restrict which terms can be targeted by an inference, the *o*λSup calculus introduces *Boolean selection functions*.

A Boolean selection function chooses *green subterms* of Boolean type, which are different than T or $\bot$ and do not occur at either side of a positive literal in a clause. It gives rise to a notion of eligibility that considers the formula structure. We call terms that can be selected *selectable subterms*. Green subterms correspond to the first-order skeleton of a higher-order term; that is, they do not occur in positions under applied variables, quantifiers, or λ-abstractions.

Definition 6.4 (Green subterms and green positions)**.** Green subterms and green positions are defined inductively as follows: t is a green subterm of t at green position ε; if t is a green subterm of u_i at green position p and f is a constant different from $\forall$ and $\exists$, then t is a green subterm of $\mathsf{f}\,\overline{u}_n$ at green position $i.p$, assuming $i \leq n$.

Example 6.5. The green subterms of the term $F\,\mathsf{a} \wedge \mathsf{p}\,(\forall\,(\lambda x.\,\mathsf{q}\,x))\,\mathsf{b}$ are the term itself, $F\,\mathsf{a}$, $\mathsf{p}\,(\forall\,(\lambda x.\,\mathsf{q}\,x))\,\mathsf{b}$, $\forall\,(\lambda x.\,\mathsf{q}\,x)$, and b.

Green positions are lifted to clauses as follows: If p is the green position of a subterm in s, and s occurs in a literal $l \in \{s \approx t,\, s \not\approx t\}$ of a clause C, the green position of the same subterm in C is denoted by $l.s.p$. *o*λSup mandates additional restrictions on the Boolean selection function: T, $\bot$, and variable-headed terms cannot be selected; for literals $s \approx t$, neither s nor t can be selected; if s (or t) contains a variable X as a green subterm and $X\,\overline{u}_n$, with $n \geq 1$, is a maximal term of C, then s (or t) cannot be selected.

Definition 6.6 (Eligibility)**.** Given a substitution σ and term order $\succ$, we say a literal l is (strictly) eligible with respect to σ in C if it is selected in C or there are no selected literals and no selected Boolean subterms in C and $\sigma(l)$ is (strictly) maximal in $\sigma(C)$ with respect to the term order. The eligible subterms of a clause C with respect to a substitution σ are inductively defined as follows: Any subterm selected by the Boolean selection function is eligible. For a strictly eligible literal $s \approx t$ with $\sigma(t) \not\succ \sigma(s)$, s is eligible. For an eligible

literal $s \not\approx t$ with $\sigma(t) \not\succ \sigma(s)$, s is eligible. If a subterm t is eligible and the head of t is not $\approx$ or $\not\approx$, all green subterms directly below the head of t are eligible. If a subterm t is eligible and t is of the form $u \approx v$ or $u \not\approx v$, then u is eligible if $\sigma(v) \not\succ \sigma(u)$ and v is eligible if $\sigma(u) \not\succ \sigma(v)$.

Example 6.7. Consider a clause $\mathsf{p} \approx \mathsf{T} \vee \mathsf{q}(\mathsf{p} \vee \mathsf{a} \approx \mathsf{b}) \approx \mathsf{T}$, literal and Boolean selection functions that select no literals and terms, respectively, and the precedence of symbols $\mathsf{T} \prec \mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{p} \prec \mathsf{q}$. As no literals and subterms are selected, the second literal is eligible, as it is maximal. This makes the term $\mathsf{q}(\mathsf{p} \vee \mathsf{a} \approx \mathsf{b})$ eligible. Further, the green subterm $\mathsf{p} \vee \mathsf{a} \approx \mathsf{b}$ is eligible. Similarly, p and $\mathsf{a} \approx \mathsf{b}$ are eligible. Lastly, as $\mathsf{a} \prec \mathsf{b}$, b is eligible.

The above definitions of green subterms and eligibility were originally introduced with $o\lambda$Sup [17]. The Boolean selection function plays a similar role as the literal selection function in standard superposition. Literal selection functions eliminate some of the nondeterminism present in the superposition calculus by focusing on selected parts of the search space. Boolean selection functions achieve the same goal, but in a different context: They eliminate nondeterminism that is not present in standard superposition, namely, the choice of subformula on which the Boolean calculus rules are to be applied. As with literal selection functions, selecting few (and smaller) subterms can give rise to fewer possible inferences and reduce clause proliferation.

This notion of eligibility opens up possibilities for reasoning with formulas that are hard to simulate with the existing superposition machinery. For example, given a formula $\varphi \longrightarrow \psi$, selecting the antecedent simulates forward reasoning, whereas selecting the consequent simulates backward reasoning. This concept of eligibility also makes it possible to restrict the proof search to a small, promising part of a formula. Note that literal selection can override Boolean selection: Selecting a literal might make some of its green subterms eligible, regardless of Boolean selection.

In our previous work [127], we left this area of new possibilities largely unexplored. We designed simple functions that selected smallest, largest, innermost, or outermost terms, but they did not impact performance much. A similar idea has been discussed in the context of the Cases(Simp) rule in Sect. 5.3. Here, we propose alternatives. Intuitively, a well-performing literal selection function might succeed at taming the combinatorial explosion if the selected literal can take part in few inferences [78]. However, Boolean selection functions introduce another factor to consider: the context in which the selected subterm occurs. This suggests the following definition:

Definition 6.8 (Contextualized Boolean selection function)**.** Let $ctx(C)$ be a function that maps a clause C to a set of green positions p such that $C|_p$ is a selectable Boolean subterm, and let $\rhd$ be a partial order on pairs of terms and green positions. The *context Boolean selection function* $Sel^{\rhd}_{ctx}(C)$ selects all terms t such that $t = C|_p$, $p \in ctx(C)$, and (t, p) is maximal with respect to $\rhd$, compared with all other pairs $(C|_{p'}, p')$, $p' \neq p$, and $p \in ctx(C)$.

In the above definition, the function ctx lets us choose the context in which the Boolean subterm appears. Then, among the terms in the chosen context, we choose the ones that are maximal with respect to $\rhd$.

Ganzinger and Stuber considered Boolean subterm selection for their extension of first-order superposition with interpreted Boolean type [69]. Unlike our calculus, their calculus only allows the selection of subterms that occur in negative green positions, defined below.

Definition 6.9 (Polarity of green positions)**.** Negative and positive green positions in a clause $C = l_1 \vee \cdots \vee l_n$ are defined inductively as follows: For each $1 \le i \le n$, the green position $l_i.s$ is positive if $l_i = s \approx \top$ and negative if $l_i = \neg s$. If p is positive (negative) and $C|_p = s\,\overline{t}_n$ where s is either $\wedge$ or $\vee$, then each $p.i, 1 \le i \le n$, is positive (negative). If p is positive and $C|_p = \neg\, s$, then $p.1$ is negative; if p is negative and $C|_p = \neg\, s$, then $p.1$ is positive. If p is positive and $C|_p = s \longrightarrow t$, then $p.1$ is negative and $p.2$ is positive; if p is negative and $C|_p = s \longrightarrow t$, then $p.1$ is positive and $p.2$ is negative.

Note that the polarity of p is undefined whenever $C|_p$ is not a green Boolean subterm or it occurs under a (dis)equivalence or an uninterpreted symbol. To assess how the function *ctx* affects performance, we use the following selection functions that consider green positions of selectable Boolean terms:

`Any`	select all green positions;
`Pos`	select all positive green positions;
`Neg`	select all negative green positions;
`Forward`	select all green positions $p = q.1$ such that $C|_q = s \longrightarrow t$;
`Backward`	select all green positions $p = q.2$ such that $C|_q = s \longrightarrow t$;
`Deep`	select all green positions of maximal length;
`Shallow`	select all green positions of minimal length.

We also introduce three partial orders for selecting subterms from a given context. For all three orders, if exactly one of the subterms has a logical head, then the subterm with the nonlogical head is larger, because logical symbols are more explosive. Otherwise, the orders use the following criteria:

$\rhd_{\text{ground}}$	If exactly one of the subterms is ground, make the ground subterm larger; otherwise, if exactly one of the subterms is of the form $s \approx t$, make this subterm larger.
$\rhd_{\text{depth}}$	If one of the subterms has larger subterm depth (longest valid green subterm position), make this subterm larger; otherwise, if one of the subterms has fewer distinct variables, make this subterm larger.
$\rhd_{\text{def}}$	If exactly one of the subterms is of the form $\mathsf{p}\,\overline{X}_n$ where $\overline{X}_n$ is a tuple of free variables, make the other subterm larger; otherwise, if exactly one of the subterms is of the form $X\,\overline{s}_n$, make the other subterm larger.

In case of a tie, the subterm with the smaller syntactic weight is made larger, and if both subterms have the same weight, the term that occurs in a position further to the left (i.e., that has a lexicographically smaller position) is made larger.

These orders follow the design principle enunciated by Hoder et al. [78] that ground or deep terms and terms with repeated variables are "less unifiable" with the similar observation for higher-order logic that reasoning about interpreted symbols or applied variables is usually explosive.

Example 6.10. Selecting the right Boolean subterm can help avoid elaborating higher-order inferences. Consider the unsatisfiable clause set consisting of $\mathsf{p}(\lambda y.\, X\,(\lambda x.\, x)\,\mathsf{a}) \longrightarrow \neg(\mathsf{p}(\lambda y.\, X\,y\,\mathsf{a})) \approx \top$, $\mathsf{p}(\lambda y.\, \mathsf{a}) \approx \top$, and $\mathsf{p}(\lambda y.\, y^{100}\,\mathsf{b}) \approx \top$, where superscript i denotes i-fold application of a unary term. Note that $\mathsf{p}(\lambda y.\, X\,(\lambda x.\, x)\,\mathsf{a})$ and $\mathsf{p}(\lambda y.\, \mathsf{a})$ have infinitely many unifiers of the form $\{X \mapsto \lambda f x.\, f^i(x)\}, i \ge 0$, whereas terms $\mathsf{p}(\lambda y.\, X\,y\,\mathsf{a})$ and $\mathsf{p}(\lambda y.\, y^{100}\,\mathsf{b})$

		Any	Pos	Neg	Forward	Backward	Deep	Shallow
TPTP	$\rhd_{\text{ground}}$	1538	1550	1547	1534	**1554**	1539	1538
	$\rhd_{\text{depth}}$	1542	1550	1528	1542	1550	1547	1535
	$\rhd_{\text{def}}$	1543	1551	1540	1540	1551	1545	1537
SH	$\rhd_{\text{ground}}$	386	379	386	386	379	**387**	**387**
	$\rhd_{\text{depth}}$	377	376	384	378	376	379	376
	$\rhd_{\text{def}}$	379	374	**387**	379	380	377	381

Figure 6.3: Impact of the Boolean selection function

have a finite CSU (in fact an MGU). When `Forward` context selection is enabled, the target of superposition inference becomes $\mathsf{p}(\lambda y.X\,(\lambda x.x)\,\mathsf{a})$, forcing computation of at least 100 unifiers (under the assumption that unifiers are returned in order of increasing i) before we get to refute $\neg(\mathsf{p}(\lambda y.y^{100}\,\mathsf{b}))$. In contrast, `Backward` context selection allows us to superpose from $\mathsf{p}(\lambda y.\,y^{100}\,\mathsf{b})$ into $\mathsf{p}(\lambda y.X\,y\,\mathsf{a})$, avoiding this explosion.

Evaluation and Discussion When the input problem is clausified using immediate clausification, almost all Boolean structure is lost. In this case, we expect Boolean selection to have a modest effect. To better assess this feature, in this evaluation we use DCI–LA from Sect. 6.4 as the baseline configuration. To avoid interference of literal and Boolean selection, we additionally forbid the literal selection function from selecting a literal if it contains a selectable Boolean subterm.

The results of evaluating 21 concrete selection functions obtained by instantiating the contextualized Boolean selection function are shown in Figure 6.3. Rows denote the partial order $\rhd$ which is used, while columns denote the function *ctx*.

On TPTP benchmarks, Boolean selection helps tame the explosion caused by dynamic clausification used as inference: All but one selection functions outperform the DCI–LA baseline of 1532 proved problems. Coming back to the problem `NUM636^2` from Sect. 6.4, using Boolean selection can reduce the number of given clause loop iterations from 294 to 71.

The results suggest that selection of term context has more impact than the partial term order. Also, the best results are obtained when a context more specific than `Any` is chosen. Remarkably, functions using the `Pos` context perform better than the ones using the `Neg` context on TPTP, but the opposite is observed on SH.

Using different Boolean selection functions yields vastly different sets of proved problems on TPTP benchmarks: In total, there are 103 problems proved by some configuration from Figure 6.3 but not by DCI–LA. However, there are only 16 such SH problems. It would seem that the advanced formula reasoning facilitated by the Boolean selection formulas is usually not required by Sledgehammer problems.

6.6 Enumerating Infinitely Branching Inferences

As an optimization and to simplify the code, Leo-III [151] and Vampire 4.4 [24] (which uses *restricted combinatory unification*, a predecessor of combinatory superposition) compute

only a finite subset of the possible conclusions of inferences that require enumerating a CSU. Not only is this a source of incompleteness, but choosing the cardinality of the computed subset is a difficult heuristic choice. Small sets can result in missing the necessary unifier, while large sets make the prover spend too long in the unification procedure, generate useless clauses, and possibly get sidetracked into wrong parts of the search space.

We propose a modification to the given clause procedure to seamlessly interleave unifier computation and proof state exploration. Given a complete unification procedure, which may yield infinite streams of unifiers, our modification fairly enumerates all conclusions of inferences relying on elements of a CSU. Under some reasonable assumptions, it behaves exactly like the standard given clause procedure on purely first-order problems. We also describe heuristics that help achieve a similar performance as when using incomplete, terminating unification procedures without sacrificing completeness.

Given that we cannot decide whether there exists a next CSU element in a stream of unifiers, a request for the next conclusion might not terminate, effectively bringing the theorem prover to a halt. Our modified given clause procedure expects the unification procedure to return a lazily computed stream [129, Sect. 4.2], where each element is either $\emptyset$ or a singleton set containing a unifier. To avoid getting stuck waiting for a unifier that may not exist, the unification procedure should return $\emptyset$ after it performs some number of operations without finding a unifier.

The complete unification procedure described in Chapter 4 returns such a stream. Other procedures, such as Huet's [82] and Jensen and Pietrzykowski's [86], can easily be adapted to meet this requirement. Based on the stream of unifiers interspersed with $\emptyset$, we can construct a stream of inferences similarly interspersed with $\emptyset$. Any finite prefixes of this stream can be computed in finite time.

To support such streams in the given clause procedure, we extend it to represent the proof state not only by the active ($\mathscr{A}$) and passive ($\mathscr{P}$) clause sets, but also by a priority queue $\mathscr{Q}$ containing the inference streams, similar to the "to do" set T present in the abstract Zipperposition loop of Waldmann et al. [171, Sect. 4]. Each stream is given a weight, and $\mathscr{Q}$ is sorted in order of increasing weight, a low weight corresponding to a high priority. When they introduced λSup, Bentkamp et al. [18] described an older version of this extension. Here we present a newer version in more detail, including heuristics to postpone unpromising streams. The pseudocode of the modified procedure is as follows:

function ExtractClause(Q, *stream*)
  maybe_clause ← pop and compute the first element of *stream*
  if *stream* is not empty **then**
    add *stream* to Q with an increased weight
  return *maybe_clause*

function FairProbe(Q, *num_oldest*)
  collected_clauses ← $\emptyset$
  oldest_streams ← pop *num_oldest* oldest streams from Q
  for *stream* in *oldest_streams* **do**
    collected_clauses ← *collected_clauses* ∪ ExtractClause(Q, *stream*)
  return *collected_clauses*

```
function HeuristicProbe(Q)
  i ← 0
  collected_clauses ← ∅
  while i < K_best and Q ≠ ∅ do
    j ← 0
    maybe_clause ← ∅
    while j < K_retry and Q ≠ ∅ and maybe_clause = ∅ do
      stream ← pop the lowest-weight stream in Q
      maybe_clause ← ExtractClause(Q, stream)
      j ← j + 1
    collected_clauses ← collected_clauses ∪ maybe_clause
    i ← i + 1
  return collected_clauses

function ForceProbe(Q)
  collected_clauses ← ∅
  while Q ≠ ∅ and collected_clauses = ∅ do
    collected_clauses ← FairProbe(Q, |Q|)
  if Q = ∅ and collected_clauses = ∅ then
    status ← Satisfiable
  else
    status ← Unknown
  return (status, collected_clauses)

function GivenClause(P, A, Q)
  i ← 0
  status ← Unknown
  while status = Unknown do
    if P = ∅ then
      (status, forced_clauses) ← ForceProbe(Q)
      P ← P ∪ forced_clauses
    else
      given ← pop a chosen clause from P and simplify it
      if given is the empty clause then
        status ← Unsatisfiable
      else
        A ← A ∪ {given}
        for stream in streams of inferences between given and other ∈ A do
          if stream is not empty then
            P ← P ∪ ExtractClause(Q, stream)
        i ← i + 1
        if i mod K_fair = 0 then
          P ← P ∪ FairProbe(Q, i/K_fair)
        else
          P ← P ∪ HeuristicProbe(Q)
  return status
```

The entry point of the above pseudocode is the function GivenClause, called with arguments $(\mathscr{P}, \mathscr{A}, \mathscr{Q})$, which are initialized as usual: All input clauses are put into $\mathscr{P}$, and $\mathscr{A}$ and $\mathscr{Q}$ are empty. In other words, the parameters P, A, and Q intuitively correspond to $\mathscr{P}, \mathscr{A}$, and $\mathscr{Q}$. Unlike in the standard given clause procedure, inference results are represented as clause streams. The first element is inserted into P, and the rest of the stream is stored in Q with some positive integer weight computed from the inference rule.

To eventually consider inference conclusions from streams in Q as given clauses, we extract elements from, or *probe*, streams and move any obtained clauses to P. Analogous to the traditional pick–given ratio [116, 143], we use a parameter K_{fair} (by default, $K_{\text{fair}} = 70$) to ensure fairness: Every K_{fair}th iteration, FairProbe probes an increasing number of oldest streams, which achieves dovetailing. In all other iterations, HeuristicProbe attempts to extract up to K_{best} clauses from the most promising streams (by default, $K_{\text{best}} = 7$). In each attempt, the most promising stream in Q is chosen. If its first element is $\emptyset$, the rest of the stream is inserted into Q and a new stream is chosen. This is repeated until either K_{retry} occurrences of $\emptyset$ have been met (by default, $K_{\text{retry}} = 20$) or the stream yields a singleton. Setting $K_{\text{retry}} > 0$ increases the chance that HeuristicProbe will return K_{best} clauses, as desired. Finally, if P becomes empty, ForceProbe searches relentlessly for a clause in Q, as a fallback. The default values for $K_{\text{fair}}, K_{\text{best}}$, and K_{retry} are chosen by informal experiments.

The function ExtractClause extracts an element from a nonempty stream not in Q and inserts the remaining stream into Q with an increased weight, calculated as follows. Let n be the number of times the stream was chosen for probing. If probing results in $\emptyset$, the stream's weight is increased by $\max\{2, n - 16\}$. If probing results in a clause C whose penalty is p, the stream's weight is increased by $p \cdot \max\{1, n - 64\}$. The penalty of a clause is a number assigned by Zipperposition based on features such as the depth of its derivation and the rules used in it. The constants 16 and 64 increase the chance that newer clause-producing streams are picked, which is desirable because their first clauses are expected to be useful.

All three probing functions are invoked by GivenClause, which contains the saturation loop. It differs from the standard given clause procedure in three ways: First, the proof state includes Q in addition to P and A. Second, new inferences involving the given clause are added to Q instead of being performed immediately. Third, a number of inferences in Q are periodically computed to fill P.

Example 6.11. In this example we simplify the notation by writing positive predicate literals $s \approx \mathsf{T}$ as s and negative predicate literals $t \not\approx \mathsf{T}$ as $\neg t$. Consider the unsatisfiable two-clause problem $\{X(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}(X\,\mathsf{a}) \vee \mathsf{p}(X\,\mathsf{a}), \neg\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a})\}$ and a selection function which selects negative literals. Let $P \mid A \mid Q$ denote the state of the given clause loop (i.e., the contents of the passive and active set and of the stream queue), and let $[a_1, a_2, \ldots]$ denote an infinite stream of elements.

The given clause loop begins in the state $X(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}(X\,\mathsf{a}) \vee \mathsf{p}(X\,\mathsf{a}), \neg\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a}) \mid \emptyset \mid \emptyset$. If the clause $\neg\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a})$ is chosen for processing, since Q is empty and no inferences with the chosen clause are possible, the state becomes $X(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}(X\,\mathsf{a}) \vee \mathsf{p}(X\,\mathsf{a}) \mid \neg\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a}) \mid \emptyset$. When the clause $X(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}(X\,\mathsf{a}) \vee \mathsf{p}(X\,\mathsf{a})$ is chosen, a new stream which enumerates results of equality resolution (on its first literal) is created. There are infinitely many conclusions of this inference, since there are infinitely many unifiers for the first literal of the form $\{X \mapsto \lambda x.\, \mathsf{f}^i\, x\}$, for $i \geq 0$. Thus, the stream is $[\{\mathsf{p}\,\mathsf{a}\}, \{\mathsf{p}(\mathsf{f}\,\mathsf{a})\}, \ldots]$, possibly with $\emptyset$s interspersed.

With the standard given clause procedure, there would have been no way to represent this infinitary result.

When the stream is created, its first element is popped and put into P. Then, based on the parameters that control inference stream probing, some number of clauses from the stream are computed and moved to P. After two iterations, the state might be $\mathsf{p}\,\mathsf{a}, \mathsf{p}(\mathsf{f}\,\mathsf{a})$, $\mathsf{p}(\mathsf{f}(\mathsf{f}\,\mathsf{a})) \mid X(\mathsf{f}\,\mathsf{a}) \not\approx \mathsf{f}(X\,\mathsf{a}) \vee \mathsf{p}(X\,\mathsf{a}), \neg\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a}) \mid [\{\mathsf{p}(\mathsf{f}^{3}\,\mathsf{a})\}, \ldots]$.

In the next iterations, some clause of the form $\mathsf{p}(\mathsf{f}^{i}\,\mathsf{a})$, where $i < 100$, is chosen, but no inferences with it can be performed. Then, the stream created in the second iteration is probed, and its results fill the set P. Ultimately, the clause $\mathsf{p}(\mathsf{f}^{100}\,\mathsf{a})$ is chosen, at which point $\bot$ is quickly derived.

GivenClause eagerly stores the first element of a new inference stream in P to imitate the standard given clause procedure. If the underlying unification procedure behaves like the standard first-order unification algorithm on higher-order logic's first-order fragment, our given clause procedure coincides with the standard one. The unification procedure described in Chapter 4 terminates on the first-order and other fragments. To avoid computing complicated unifiers eagerly, it immediately returns $\emptyset$ for a problem that does not belong to one of the fragments that admit efficient unifier computation.

The design of our given clause procedure was guided by folklore knowledge about higher-order theorem proving. First, in our experience most steps in long higher-order proofs involve first-order literals. The unification procedure and inference scheduling ensure that first-order inference conclusions are put in the proof state as early as possible. Second, some inference rules are expected to be largely useless. We initialize the stream penalty differently for each rule, allowing old streams of more useful inferences to be queried before newly added, but potentially less useful streams. Finally, if we use a unification procedure that has aggressive redundancy elimination, we will often find the necessary unifier within the first few unifiers returned. Similarly, if a stream keeps returning $\emptyset$, it is likely that it is blocked in a nonterminating computation and should be ignored. Our heuristics to increase the stream penalties take these observations into account.

Evaluation and Discussion The evaluation of our unification algorithm in Sect. 4.8 shows that Zipperposition is the only competing higher-order prover that proves all Church numeral problems from the TPTP, never spending more than 5 s on a problem. On these hard unification problems, the stream system allows the prover to explore the proof state lazily.

Consider the TPTP problem `NUM800^1`, which requires finding a function F such that $F\,\mathsf{c}_1\,\mathsf{c}_2 \approx \mathsf{c}_2 \wedge F\,\mathsf{c}_2\,\mathsf{c}_3 \approx \mathsf{c}_6$, where c_n abbreviates the Church numeral for n, $\lambda s\,z.\, s^n\,z$. To prove the problem, it suffices to take F to be the multiplication operator $\lambda x\,y\,s\,z.\, x\,(y\,s)\,z$. However, this unifier is only one out of many available for each occurrence of F.

To evaluate our unification algorithm (using a somewhat different evaluation setup and an older version of Zipperposition), we also compared a complete, nonterminating variant of the unification procedure and a pragmatic, terminating variant. The pragmatic variant was used directly—all the inference conclusions were put immediately in P, bypassing Q. The complete variant, which relies on possibly infinite streams and is much more prolific, proved only 15 problems fewer than the most competitive pragmatic variant. Furthermore, it proved 19 problems not proved by the pragmatic variant. This shows that our

given clause procedure, with its heuristics, allows the prover to defer exploring less promising branches of the unification and uses the full power of a complete higher-order unifier search to solve unification problems that cannot be proved by a restricted procedure.

The parameters K_{fair}, K_{retry}, and K_{best} can greatly influence the behavior of the given clause procedure, even when the same unification procedure is used. Figure 6.4 presents the effects of these parameters on TPTP and SH. Selecting a low number of best clauses seems to perform well on both benchmark sets. However, on SH benchmarks, which mostly require first-order unifiers, visiting older streams should be delayed a lot.

As with Boolean selection functions, changing these three parameters causes a substantial difference in the set of proved problems. For example, the configuration that performs the worst on TPTP benchmarks proves 12 problems that the configuration performing the best on TPTP cannot prove; moreover, there are 29 TPTP problems that are proved by some set of parameters other than $K_{\text{fair}} = K_{\text{best}} = 16, K_{\text{retry}} = 2$. On SH, these effects are much weaker; most reasonable combinations of parameters perform similarly.

Among the competing higher-order provers, only Satallax uses infinitely branching calculus rules. It maintains a queue of "commands" that contain instructions on how to create a successor state in the tableau. One command describes an infinite enumeration of all closed terms of a given function type. Each execution of this command makes progress in the enumeration. In contrast to computing inferences using streams representing elements of CSU, each command execution is guaranteed to make progress in enumerating the next closed functional term, so there is no need to ever return $\emptyset$.

6.7 Controlling Prolific Rules

To support higher-order features such as function extensionality and quantification over functions, many refutationally complete calculi employ highly prolific rules. For example, λSup includes a FluidSup rule [18] that very often applies to two clauses if one of them contains a term of the form $F\,\overline{s}_n$, where $n > 0$. This rule is inherited in the successor of the calculus, $o\lambda$Sup, that we use in this chapter. We describe three mechanisms to keep rules like these under control.

First, *we limit applicability of the prolific rules.* In practice, it often suffices to apply prolific higher-order rules only to initial or shallow clauses—clauses with a shallow derivation depth. Thus, we added an option to forbid the application of a rule if the derivation depth of any premise exceeds a limit.

Second, *we penalize the streams of expensive inferences.* The weight of each stream is given an initial value based on characteristics of the inference premises such as their derivation depth. For prolific rules such as FluidSup, we increment this value by a parameter K_{incr}. Weights for less prolific variants of this rule, such as DupSup [18], are increased by a fraction of K_{incr} (e.g., $\lfloor K_{\text{incr}}/3 \rfloor$).

Third, *we defer the selection of prolific clauses.* To select the given clause, most saturating provers evaluate clauses according to some criteria and choose the clause with the lowest evaluation. To make this choice efficient, passive clauses are organized into a priority queue ordered by their evaluations. Like E, Zipperposition maintains multiple queues, ordered by different evaluations, that are visited in a round-robin fashion. It also uses E's two-layered evaluation, a variant of which has recently been implemented in Vampire [70]. The two layers are *clause priority* and *clause weight.* Clauses with higher priority are pre-

		K_{fair} = 2			K_{fair} = 16			K_{fair} = 128		
		K_{retry} = 2	16	128	2	16	128	2	16	128
K_{best}	2	1643	1645	1645	1661	1661	1658	1669	1664	1664
	16	1647	1646	1609	**1670**	1654	1602	1665	1659	1597
	128	1646	1644	1583	1661	1656	1577	1665	1658	1576

(a) TPTP benchmarks

		K_{fair} = 2			K_{fair} = 16			K_{fair} = 128		
		K_{retry} = 2	16	128	2	16	128	2	16	128
K_{best}	2	460	455	454	465	463	458	466	461	461
	16	458	453	445	464	459	441	**468**	459	442
	128	456	452	430	465	458	428	**468**	459	425

(b) SH benchmarks

Figure 6.4: Impact of the stream enumeration parameter

ferred, and the weight is used for tie-breaking. Intuitively, the first layer crudely separates clauses into priority classes, while the second one uses heuristic weights to prefer clauses within a priority class. To control the selection of prolific clauses, we introduce new clause priority functions that take into account features specific to higher-order clauses.

The first new priority function, `PreferHOSteps` (`PHOS`), assigns a higher priority if rules specific to higher-order superposition calculi were used in the clause derivation. Since most of the other clause priority functions tend to defer higher-order clauses, having a clause queue that prefers them might be useful to find some proof more efficiently. A simpler function, which prefers clauses containing λ-abstractions, is `PreferLambda` (`PL`).

`PreferHOSteps` separates clauses created using first- and higher-order inference rules crudely. However, within higher-order inference rules there are the ones which make clauses simpler and are thus more preferable. An example of such a rule is

$$\frac{C \vee s \approx t}{C \vee s\overline{X}_n \approx t\overline{X}_n}\ \textsc{ArgCong}$$

where s is of the type $\alpha_1 \to \cdots \to \alpha_k \to \beta$, β is a base type, $n \leq k$, free variables $\overline{X}_n$ are fresh, and literal $s \approx t$ is strictly eligible (for paramodulation). When $n = k$, in most cases, the resulting clause has a first-order literal $s\overline{X}_n \approx t\overline{X}_n$ in place of the literal $s \approx t$ of functional type, which usually makes the clause more useful. To prefer clauses that are only mildly higher-order, we designed the function `PreferEasyHO` (`PEHO`). It prefers clauses that are the result of ArgCong, have equations between terms of functional type or between higher-order patterns, or have literals containing logical symbols, in that order of priority.

A higher-order inference that applies a complicated substitution to a clause is usually followed by a $\beta\eta$-reduction step. If $\beta\eta$-reduction greatly reduces the size of a clause, it is likely that this substitution simplifies the clause (e.g., by removing a variable's arguments). The new priority function `ByNormalizationFactor` (`BNF`) is designed to exploit this observation. It prefers clauses that were produced by $\beta\eta$-reduction, and among those it prefers the ones with larger size reductions.

Another new priority function is `PreferShallowAppVars` (`PSAV`). This prefers clauses with lower depths of the deepest occurrence of an applied variable—that is, $C[X\,\mathtt{a}]$ is preferred over $C[\mathtt{f}(X\,\mathtt{a})]$. The intuition is that applying a substitution to an applied variable often reduces the variable to a term with a constant head, yielding a less explosive clause, and the gain is greater for variables closer to the top level. Among the functions that rely on properties of applied variables, we implemented `PreferDeepAppVars` (`PDAV`), which returns the priority opposite of `PSAV`, and `ByAppVarNum` (`BAVN`), which prefers clauses with fewer occurrences of applied variables.

Evaluation and Discussion In the base configuration (*base*), Zipperposition visits several clause queues. The configuration uses queues that prefer the clauses that stem from the conjecture, the ones that have at least one positive literal, the ones that have been moved from the active to the passive set, and so on. One of the queues uses the constant priority function `ConstPrio` (`CP`), meaning that it assigns the same priority to every clause. As this queue is the most often visited one in *base*, changing its priority function should affect the result noticeably. To evaluate the new priority functions, we replaced `CP` with

	CP	BAVN	PL	PSAV	PHOS	PEHO	BNF	PDAV
TPTP	1635*	**1640**	1604	1635	1609	1617	1575	1533
SH	**452***	451	417	450	439	407	411	302

Figure 6.5: Impact of the priority function

	∞	16	8	4	2	1
TPTP	**1635***	1625	1632	1629	1628	1618
SH	**452***	438	435	439	435	440

Figure 6.6: Impact of the FluidSup weight increment K_{incr}

one of the new functions in this queue, leaving the clause weight intact. The results are shown in Figure 6.5.

Even though the constant priority function achieves a remarkable performance, the new priority functions are useful additions to the prover's repertoire: 37 additional TPTP problems and 17 additional SH problems can be proved when some nonconstant priority is used. The generally average-performing PEHO function can prove nine problems not proved with any other priority function on TPTP (and one on SH). Globally, 24 TPTP problems and six SH problems can be proved exclusively using one particular priority function.

Although it is necessary for refutational completeness, the FluidSup rule is disabled in *base* because it is so explosive and so seldom useful. To test whether increasing inference stream weights makes a difference on the success rate, we tried enabling FluidSup with different weight increments K_{incr} for FluidSup inference queues. The results are shown in Figure 6.6. As expected, using a low increment with FluidSup is detrimental on TPTP. On this benchmark set, 16 additional problems can be proved when FluidSup is enabled. The penalty mostly affects only proving time: All but two of these problems were proved by using at least three different values of K_{incr}. On SH problems, the best result is obtained when the rule is disabled as well. Unexpectedly, the next best result is obtained when $K_{\text{incr}} = 1$.

6.8 Controlling the Use of Backends

Cooperation with efficient off-the-shelf first-order theorem provers is an essential feature of higher-order theorem provers such as Leo-III [151, Sect. 4.4] and Satallax [39]. Those provers invoke first-order backends repeatedly during a proof attempt and spend a substantial amount of time in backend collaboration. Since λSup generalizes a highly efficient first-order calculus, we expect that future efficient λSup implementations will not benefit much from backends. Nevertheless, experimental provers such as Zipperposition can still gain a lot. We present some techniques for controlling the use of backends.

In his thesis [151, Sect. 6.1], Steen extensively studies the effects of using different first-order backends on the performance of Leo-III. His results suggest that adding only one

	–Ehoh	0.1	0.25	0.5	0.75
TPTP	1635⋆	**1981**	1980	1979	1972
SH	452⋆	606	**608**	600	592

Figure 6.7: Impact of the backend invocation point K_{time}

	–Ehoh	lifting	SKBCI	omitted
TPTP	1635⋆	**1980**	1877	1866
SH	452⋆	**608**	577	566

Figure 6.8: Impact of the method used to translate λ-abstractions

backend already substantially improves the performance. To reduce the effort required for integrating multiple backends, we chose Ehoh, an extension of E [147] that supports λfSup, as our single backend. In Chapter 3, we described Ehoh in detail. On the one hand, Ehoh provides the efficiency of E while easing the translation from full higher-order logic—the only missing syntactic feature is λ-abstraction. On the other hand, Ehoh's higher-order reasoning capabilities are limited. Its unification algorithm is essentially first-order, and it cannot synthesize λ-abstractions.

In a departure from Leo-III and other cooperative provers, instead of regularly invoking the backend, we invoke it at most once during a run of Zipperposition. This is because most competitive higher-order provers, including Zipperposition, use a portfolio mode in which many configurations are run for a short time, and we want to leave enough time for native higher-order reasoning. Moreover, multiple backend invocations tend to be wasteful, because currently each invocation starts with no knowledge of the previous ones.

Only a carefully chosen subset of the available clauses are translated and sent to Ehoh. Let I be the set of clauses representing the input problem. Given a proof state, let M denote the union of the current active and passive sets, and let M_{ho} denote the subset of M that contains only clauses that were derived using at least one λSup rule not present in regular superposition. We order the clauses in M_{ho} by increasing derivation depth, using syntactic weight to break ties. Then we choose all clauses in I and the first K_{size} clauses from M_{ho} for use with the backend reasoner. We leave out clauses in $M \setminus (I \cup M_{\text{ho}})$ because Ehoh can rederive them. We also expect large clauses with deep derivations to be less useful.

The remaining step is the translation of λ-abstractions. We implemented two translation methods: λ-lifting [87] and SKBCI combinators [165]. For SKBCI, we omit the combinator definition axioms, because they are very explosive [25]. A third mode simply omits clauses containing λ-abstractions.

Evaluation and Discussion In Zipperposition, we can adjust the CPU time allotted to Ehoh, Ehoh's own parameters, the point when Ehoh is invoked, the number K_{size} of selected clauses from M_{ho}, and the λ translation method. We fix the time limit to 3 s, use Ehoh in *autoschedule* mode, and focus on the last three parameters. In *base*, collaboration with Ehoh is disabled (labeled –Ehoh).

Ehoh is invoked after $K_{\text{time}} \cdot t$ CPU seconds, where $0 \leq K_{\text{time}} < 1$ and t is the total CPU time allotted to Zipperposition. Figure 6.7 shows the effect of varying K_{time} when $K_{\text{size}} = 32$ and λ-lifting is used. The evaluation confirms that using a highly optimized backend such as Ehoh greatly improves the performance of a less optimized prover such as Zipperposition. The figure indicates that it is preferable to invoke the backend early. We have observed that if the backend is invoked late, small clauses with deep derivations

	–Ehoh	16	32	64	128	256	512
TPTP	1635*	**1985**	1980	1978	1968	1968	1919
SH	452*	606	**608**	600	598	596	589

Figure 6.9: Impact of the number of selected clauses K_{size}

tend to be present. These clauses might have been used to delete important shallow clauses already. But due to their derivation depth, they will not be sent to Ehoh. In such situations, it is better to invoke the backend before the important clauses are deleted.

Figure 6.8 quantifies the effects of the three λ-abstraction translation methods. We fixed $K_{\text{time}} = 0.25$ and $K_{\text{size}} = 32$. The clear winner is λ-lifting. SKBCI combinators perform slightly better than omitting clauses containing λ-abstractions.

Figure 6.9 shows the effect of K_{size} on performance, with $K_{\text{time}} = 0.25$ and λ-lifting. Including a small number of higher-order clauses with the lowest weight performs better than including a large number of such clauses.

6.9 Comparison with Other Provers

Different choices of parameters lead to noticeably different sets of proved problems. In an attempt to use Zipperposition 2 to its full potential, we created a portfolio mode that runs up to 50 configurations in parallel during the allotted time. The portfolio was designed to solve as many problems as possible from the TPTP benchmark set. To provide some context, we compare Zipperposition 2 with the following versions of all other higher-order provers that competed at CASC in 2020: CVC4 1.9 [12], Leo-III 1.5.6 [153], Satallax 3.5 [39], and Vampire 4.5.1 [25]. The provers were run using the same parameters as in CASC, but with updated executables. Note that Vampire's higher-order schedule is optimized for running on a single core. We also include Ehoh 2.7, the first version of this prover to syntactically support full higher-order logic, including λ-abstractions. Semantically, Ehoh 2.7 is arguably the weakest among the listed provers: It simply performs o-RW rewriting described in Sect. 6.3 followed by λ-lifting before it applies λfSup [15] on the preprocessed problem. It is a conservative extension of Ehoh presented in Chapter 3, and serves as a baseline for comparison with λE introduced in Chapter 7.

We use the same benchmark sets as elsewhere in this chapter. To imitate the setup of the 2020 edition of CASC, we use a 120 s wall-clock limit and a 960 s CPU limit. We carried out our evaluation on the 8-core CPU nodes that were used for CASC in 2020. We also ran Zipperposition in uncooperative mode, in which its collaboration with a backend is disabled. Figure 6.10 summarizes the results.

The evaluation results corroborate the CASC results. They also show that Zipperposition outperforms all other provers on SH benchmarks. This confirms our hypothesis that $o\lambda$Sup is a suitable basis for automatic higher-order reasoning. Further confirmation is provided by the success rate of Zipperposition's uncooperative version: Even without backend, Zipperposition is substantially better than all other provers on TPTP benchmarks, and it matches the performance of the top contenders on SH. On the other hand, the increase in performance due to the addition of an efficient backend suggests that the implemen-

	TPTP	SH
CVC4	1816	587
Ehoh	1980	676
Leo-III	2122	616
Satallax	2175	588
Vampire	2072	660
Zipperposition-uncoop	2311	652
Zipperposition	**2412**	**715**

Figure 6.10: Comparison of competing higher-order theorem provers

tation of this calculus in a modern first-order superposition prover such as E or Vampire would achieve even better results. In Chapter 7 we describe how a version of $o\lambda$Sup is implemented in E.

We believe that there are still techniques inspired by tableaux, SAT solving, and SMT solving that can be adapted and integrated in saturation provers. In particular, there are still 25 TPTP problems and 17 SH problems that can be proved by other provers but not by Zipperposition.

6.10 Discussion and Conclusion

Back in 1994, Kohlhase [96, Sect. 1.3] was optimistic about the future of higher-order automated reasoning:

> The obstacles to proof search intrinsic to higher-order logic may well be compensated by the greater expressive power of higher-order logic and by the existence of shorter proofs. Thus higher-order automated theorem proving will be practically as feasible as first-order theorem proving is now as soon as the technological backlog is made up.

For higher-order superposition, the backlog consisted of designing calculus extensions, heuristics, and algorithms that mitigate its weaknesses. In this chapter, we presented such enhancements, justified their design, and evaluated them. We explained how each weak point in the higher-order proving pipeline could be improved, from preprocessing to reasoning about formulas, to delaying unpromising or explosive inferences, to invoking a backend. Our evaluation indicates that higher-order superposition is now the state of the art in higher-order reasoning.

Higher-order extensions of first-order superposition have been considered by Bentkamp et al. [15, 18] and Bhayat and Reger [24, 25]. They introduced proof calculi, proved them refutationally complete, and suggested optional rules, but they hardly discussed the practical aspects of higher-order superposition. Extensions of SMT are discussed by Barbosa et al. [11]. Bachmair and Ganzinger [6], Manna and Waldinger [110], and Murray [122] have studied nonclausal resolution calculi.

In contrast, there is a vast literature on practical aspects of first-order reasoning using superposition and related calculi. The literature evaluates various procedures and tech-

niques [79, 136], literal and term order selection functions [78], and clause evaluation functions [70, 148], among others. Our work joins the select club of publications devoted to practical aspects of higher-order reasoning [21, 60, 152, 174].

The work presented in this chapter proved invaluable to reaching the last stop on the road to the implementation of full higher-order logic in a state-of-the-art first-order prover. We used the results and experience we gained with this work to choose a set of well-performing easy-to-implement calculus extensions and heuristics. We implemented them in Ehoh, obtaining a prover called λE. This new prover substantially increases the higher-order reasoning capabilities of Ehoh. Even without all of the techniques described in this chapter, it outperforms all higher-order provers except for Zipperposition. We give a detailed description of λE, as well as the reasoning behind all design decisions we took in Chapter 7.

7

Extending a Brainiac Prover to Higher-Order Logic

Joint work with
Jasmin Blanchette and Stephan Schulz

The automatic discharge of tedious subgoals is high on the wishlist of many users of proof assistants. Some proof assistants discharge such goals by translating them to first-order logic and invoking an efficient prover on them, but much is lost in translation. As an alternative, we propose to extend a first-order prover with native support for higher-order features. Building on our extension of E to λ-free higher-order logic, we extended E to full higher-order logic. The resulting prover is the strongest one on benchmarks coming from a proof assistant, and the second strongest on TPTP benchmarks.

In this work I designed, implemented, and evaluated all changes to term representation, algorithms, and indexing data structures. Jasmin Blanchette did the daily supervision. Stephan Schulz provided the necessary E expertise.

7.1 Introduction and Background

In Chapter 3 of this thesis we introduced Ehoh, a rather conservative extension of state-of-the-art first-order prover to a fragment of higher-order logic devoid of λ-abstraction. This extension gave us a feeling for the difficulties that might be encountered on the way to full higher-order logic. In the chapters that precede this one, we discussed many ways in which those difficulties can be overcome. In this chapter, we fulfill the promise given in the beginning of the thesis: We present the extension of Ehoh to full higher-order logic using incomplete variants of λ-superposition. This prover is called λE.

The λ-superposition calculi were previously implemented in Zipperposition, and extensive experiments with various heuristic choices have been performed (Chapter 6). In λE's implementation we used this experience to choose a set of effective rules that could easily be retrofitted into an originally first-order prover. Another principle that guided the design of λE was *gracefulness*: We made sure that our changes do not impact the strong first-order performance of E and λ-free higher-order performance of Ehoh.

One of the main challenges we faced was retrofitting λ-terms in Ehoh's term representation (Sect. 7.2). Furthermore, Ehoh's main inference engine was designed with the assumption that it will be used with inferences that compute an MGU. We implemented a higher-order unification procedure (Chapter 4) that can return multiple unifiers (Sect. 7.3) and integrated it in the inference engine. Finally, we extended and adapted the rules of superposition calculus, resulting in an incomplete, pragmatic variant of λ-superposition (Sect. 7.4).

We evaluated λE on a selection of proof assistants benchmarks as well as all higher-order theorems in the TPTP library [157] (Sect. 7.5). We found that λE clearly outperformed Ehoh on all benchmarks. It outperformed all other higher-order provers on the proof assistant benchmarks; on the TPTP benchmarks it ended up second only to the cooperative version of Zipperposition, which employs Ehoh as a backend. An arguably fairer comparison without the backend puts λE in first place for both benchmark suites. We also compared the performance of λE with E on first-order problems and found that no overhead has been introduced by the extension to higher-order logic.

Background The logic that λE targets is the monomorphic higher-order logic described in Sect. 2.3. We reuse all the notions from this section, with the corresponding notations. As in the previous chapter, we simplify the notation by writing predicate literals in unencoded form: Positive literal $s \approx \mathsf{T}$ is written as s and negative literal $s \not\approx \top$ is written as $\neg s$. This chapter discusses three tightly related provers E, Ehoh, and λE, which are disambiguated as follows:

- E is a state-of-the-art first-order prover based on superposition. It is described in Sect. 2.5.7.
- Ehoh is an extension of E to support λ-free higher-order logic. Chapter 3 is dedicated to Ehoh.
- λE further builds on Ehoh to support full higher-order logic. It is the latest prover described in this chapter.

7.2 Terms

E has been designed around perfect term sharing [109], a design that we carried on to Ehoh and λE: Any two structurally identical terms are guaranteed to be the same object in memory. This is achieved through term *cells*, which represent individual terms. Each cell has (among other fields) (1) `f_code`, an integer corresponding to the symbol at the head of the term (negative if the head is a free variable, positive otherwise); (2) `num_args`, corresponding to the number of arguments applied to the head; and (3) `args`, a size-`num_args` array of pointers to argument terms. We use the first-order notation $\mathsf{f}(s_1, \ldots, s_n)$ to denote a cell whose `f_code` corresponds to `f`, `num_args` equals n, and `args` points to the cells for $s_1, \ldots s_n$.

Ehoh represents λ-free higher-order terms using flattened, spine notation (Sect. 3.3). Thus, the terms `f`, `fa`, and `fab` are represented by the cells `f`, `f(a)`, and `f(a,b)`, respectively. To ensure free variables are perfectly shared, Ehoh treats applied free variables differently: Arguments are not applied directly to a free variable, but using an internal symbol `@` of variable arity. For example, the term $X\,\mathsf{a}\,\mathsf{b}$ is represented by the cell $\mathtt{@}(X, \mathsf{a}, \mathsf{b})$. This ensures that two different occurrences of the free variable X correspond to the same object, which makes substitutions more efficient.

Representation of λ-Terms To support full higher-order logic, Ehoh's λ-free cell data structure had to be extended to support the λ binder. We use the locally nameless representation [45] for this purpose: De Bruijn indices represent (possibly loose) bound variables, whereas we keep the current representation for free (and applied) variables.

Extending the term representation of Ehoh with a new term kind involves intricate manipulation of the cell data structure. De Bruijn indices must be represented as other cells with either a negative or a positive `f_code`. However, this has to be done in such a way that a De Bruijn index can never be mistaken for a function symbol or a variable.

Other than possibly being instantiated during β-reduction, De Bruijn indices mostly behave as constants. Therefore, we decided to represent De Bruijn indices using positive `f_codes`: The De Bruijn index of value i will have i as the `f_code`. To ensure De Bruijn indices are not mistaken for function symbols, we use the `properties` bitfield of the cell, which holds precomputed properties of the cell. We introduce the property `IsDBVar` to denote that the cell represents a De Bruijn index. Any attempt to create a De Bruijn index is performed through a dedicated library function that sets the `IsDBVar` property for every term it returns. When given the same De Bruijn index and type, this function is guaranteed to always return the same object. Finally, we have guarded all the functions and macros that manipulate function codes with the check if the property `IsDBVar` is set. To ensure perfect sharing of every occurrence of De Bruijn indices, arguments to De Bruijn indices are applied like for free variables, using `@`.

Extending cells to support λ-abstraction is easier. Each λ-abstraction has the distinguished function code `LAM` as the head symbol and two arguments: (1) a De Bruijn index 0 of the type of the abstracted variable; (2) the body of the λ-abstraction. Consider the term $\lambda x.\lambda y.\mathsf{f}\,x\,x$, where both x and y have the type ι. This term is represented as $\lambda\lambda\,\mathsf{f}\,\mathbf{1}\,\mathbf{1}$ in locally nameless representation, where bold numbers represent De Bruijn indices. In λE, the same term is represented by the cell $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathsf{f}(\mathbf{1}, \mathbf{1})))$, where all De Bruijn variables have type ι.

The first argument of LAM is redundant, since it can be deduced from the type of the λ-abstraction. However, basic λ-term manipulation operations often require access to this term. We store it explicitly to avoid creating it repeatedly.

Efficient β-Reduction Terms are stored in $\beta\eta$-reduced form. As these two reductions are performed very often, they ought to be efficient. λE performs β-reduction by reducing the leftmost outermost β-redex first. To represent β-redexes, it uses the @ symbol. Thus, the term $(\lambda x.\, \lambda y.\, (x\, y))\mathtt{f}\,\mathtt{a}$ is represented by $@(\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, @(\mathbf{1}, \mathbf{0}))), \mathtt{f}, \mathtt{a})$. Another option would have been to add arguments applied to λ-term directly to the λ representation (as in $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, @(\mathbf{1}, \mathbf{0})), \mathtt{f}, \mathtt{a})$), but this would break the invariant that LAM has two arguments. Furthermore, replacing free variables with λ-abstractions (e.g., replacing X with $\lambda x.\, x$ in $@(X, \mathtt{a})$) would require additional normalization.

A term is β-reduced as follows: When a cell of the form $@(\mathtt{LAM}(\mathbf{0}, s), t)$ is encountered, the field binding (normally used to record the substitution for a free variable) of the cell $\mathbf{0}$ is set to t. Then s is traversed to instantiate every loose occurrence of $\mathbf{0}$ in s with binding, whose loose bound De Bruijn indices are shifted by the number of λ binders above the occurrence of $\mathbf{0}$ in s [90]. Next, the same procedure is performed on the resulting term and its subterms, in leftmost outermost fashion.

λE's basic β-normalization mechanism works in this way, but it features a few optimizations. First, λE recognizes terms of the form $(\lambda \overline{x}_n.\, s)\, \overline{t}_n$ and performs parallel replacement of the bound variables x_i with t_i. Since intermediate terms are not constructed, this reduces the number of recursive function calls and calls to the cell allocator.

Second, in line with the gracefulness principle, we wanted λE to incur little (if any) overhead on first-order problems and to excel on higher-order problems with a large first-order component. If β-reduction is implemented naively, finding a β-redex involves traversing the entire term. On purely first-order terms, β-reduction is then a complete waste of time.

To avoid this, we use Ehoh's perfectly shared terms and their properties field. We introduce the property HasBetaReducibleSubterm, which is set if a cell is β-reducible. Whenever a new cell that contains a β-reducible term as a direct subterm is shared, the property is set. Setting of the property is inductively continued when further superterms are shared. For example, in the term $t = \mathtt{f}\,\mathtt{a}\,(\mathtt{g}\,((\lambda x.\, x)\,\mathtt{a}))$, the cells for $(\lambda x.\, x)\,\mathtt{a}$, $\mathtt{g}\,((\lambda x.\, x)\,\mathtt{a})$, and t itself have the property HasBetaReducibleSubterm set. When it needs to find β-reducible subterms, λE will visit only the cells with this property set. This further means that on first-order subterms, a single bit masking operation is enough to determine that no subterm should be visited.

Along similar lines, we added a property HasDBSubterm that caches whether the cell contains a De Bruijn subterm. This makes instantiating De Bruijn indices during β-normalization faster, since only the subterms that contain De Bruijn indices must be visited. Similarly, some other operations such as shifting De Bruijn indices or determining whether a term is closed (i.e., it contains no loose bound variables) can be sped up or even avoided if the term is first-order.

Efficient η-Reduction The term $\lambda x.\, s\, x$ is η-reduced to s whenever x is not a loose bound variable in s. Caching the property of η-reducibility of the term is not as beneficial

as the one for β-reducibility, because checking this property at the term's top level is done in $O(|s|)$, compared with constant time for β-reducibility. However, we use the observation that a term cannot be η-reduced if it has no λ-abstraction subterms and introduce a property `HasLambda` that notes the presence of λ-abstraction in a term. Only terms with this property are visited during η-reduction.

λE performs parallel η-reduction: It recognizes terms of the form $\lambda\overline{x}_n.\, s\,\overline{x}_n$ such that none of the x_i occurs loose bound in s. If done naively, reducing terms of this kind requires up to n traversals of s to check if each x_i occurs in s. In λE, exactly one traversal of s is required.

More specifically, when η-reducing a cell $\mathtt{LAM}(\mathbf{0}, s)$, λE considers all λ binders in s as well. In general, the cell will be of the form $\mathtt{LAM}(\mathbf{0}, \ldots, \mathtt{LAM}(\mathbf{0}, t)\ldots)$, where t is not a λ-abstraction, and l is the number of `LAM` symbols above t. Then λE breaks down the body t into a maximal decomposition $u(\boldsymbol{n}-\mathbf{1})\ldots\mathbf{1}\,\mathbf{0}$. If $n = 0$, the cell is not η-reducible. Otherwise, u is traversed to determine the minimal index $\boldsymbol{j}$ of a loose De Bruijn index, taking $\boldsymbol{j} = \infty$ if no such index exists. Then the $k = \min\{j, l, n\}$ rightmost outermost λ binders in $\mathtt{LAM}(\mathbf{0}, \ldots, \mathtt{LAM}(\mathbf{0}, t)\ldots)$ can be removed and t can be replaced by the variant of $u(\boldsymbol{n}-\mathbf{1})\ldots(\boldsymbol{k}+\mathbf{1})\,\boldsymbol{k}$ obtained by shifting the loose De Bruijn indices down by k.

To better understand this convoluted De Bruijn arithmetic, consider the term $\lambda x.\, \lambda y.\, \lambda z.\, \mathtt{f}\,x\,x\,y\,z$. This term is represented by the cell $\mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathtt{LAM}(\mathbf{0}, \mathtt{f}(\mathbf{2}, \mathbf{2}, \mathbf{1}, \mathbf{0}))))$. λE splits $\mathtt{f}(\mathbf{2}, \mathbf{2}, \mathbf{1}, \mathbf{0})$ into two parts: $u = \mathtt{f}\,\mathbf{2}$ and the arguments $\mathbf{2}, \mathbf{1}, \mathbf{0}$. Since the minimal index in u is $\mathbf{2}$, we can omit the De Bruijn indices $\mathbf{1}$ and $\mathbf{0}$ and their λ binders, yielding the η-reduced cell $\mathtt{LAM}(\mathbf{0}, \mathtt{f}(\mathbf{0}, \mathbf{0}))$.

The use of the `HasLambda` property ensures that η-reduction is not tried on first-order or λ-free higher-order terms, whereas parallel η-reduction both speeds up η-reduction and avoids creating a linear number of intermediate terms. For finding the minimal loose De Bruijn index, optimizations such as the `HasDBSubterm` property are used.

Representation of Boolean Terms E represents Boolean terms using term cells whose `f_codes` correspond to internal codes reserved for logical symbols. Quantified formulas are represented by cells in which the first argument is the quantified variable and the second one is the body of the quantified formula. For example, the term $\forall x.\, \mathtt{p}\,x$ corresponds to the cell $\forall(X, \mathtt{p}(X))$, where X is a regular free variable.

This representation is convenient for parsing formulas and clausification, which is what E uses it for, but it causes α-normalization issues during the actual proof search: In full higher-order logic, Boolean terms can appear as subterms in clauses, as in $\mathtt{q}(X) \vee \mathtt{p}(\forall(X, \mathtt{r}(X)))$; instantiating X in the first literal should not influence X in the second literal.

To avoid this issue, in λE we use λ binders to represent quantified formulas. Thus, $\forall x.\, s$ is represented by $\forall(\lambda x.\, s)$. Quantifiers are then unary symbols that do not directly bind the variables but delegate this responsibility to a λ-abstraction. Since λE represents bound variables using De Bruijn indices, this solves the α-conversion issues. However, this solution is incompatible with thousands of decades-old lines of clausification code that assumes the E representation of quantified formulas. Therefore, λE converts quantified formulas only after clausification, for Boolean terms that appear in a higher-order context (e.g., as argument to a function symbol).

New Term Orders The λ-superposition calculus is parameterized by a term order that is used to break symmetries in the search space. We implemented the versions of the Knuth–Bendix order (KBO) and lexicographic path order (LPO) for higher-order terms with λ-abstractions described by Bentkamp et al. [17]. These orders encode λ-terms as first-order terms and then invoke the standard KBO or LPO. For efficiency, we implemented separate KBO and LPO functions that compute the order directly, intertwining the encoding and the order computation.

Ehoh cells contain a `binding` field that can be used to store the substitution for a free variable. Substitutions can then be applied by following the `binding` pointers, replacing each free variable with its instance. Thus, when Ehoh needs to perform a KBO or LPO comparison of an instantiated term, it needs only follow the `binding` pointers. In full higher-order logic, however, instantiating a variable can trigger a series of $\beta\eta$-reductions, changing the shape of the term dramatically. To prevent this, λE computes the $\beta\eta$-reduced instances of the terms before comparing them using KBO or LPO.

7.3 Unification, Matching, and Term Indexing

Standard superposition crucially depends on the concept of an MGU. In higher-order logic, such a unifier does not in general exist, and the concept is replaced by that of a complete set of unifiers (CSU), which may be infinite. In Chapter 4, we described an efficient procedure to enumerate a CSU for a term pair. It is implemented in Zipperposition, together with some extensions to term indexing. In λE, we further improve the performance of this procedure by implementing a terminating, incomplete variant. We also introduce a new indexing data structure.

7.3.1 The Unification Procedure

The unification procedure works by maintaining a list of unification pairs to be solved. After choosing a pair, it first normalizes it by β-reducing and instantiating the heads of both terms in the pair. Then, if either head is a variable, it computes an appropriate binding for this variable, thereby approximating the solution.

Unlike in first-order and λ-free higher-order unification, in the full higher-order case there may be many bindings that lead to a solution. To reduce this mostly blind guessing of bindings, the procedure features support for *oracles* (Sect. 4.3). These are procedures that solve the unification problem for a subclass of higher-order terms on which unification is decidable and, in the case of λE, unary. Oracles help increase performance, avoid nontermination, and avoid redundant bindings.

In Chapter 4, the unification procedure is described as a transition system. In λE, the procedure is implemented nonrecursively, and the unifiers are enumerated using an iterator object that encapsulates the unifier search state. The iterator consists of five fields:

1. *constraints*, which holds the unification constraints

2. *bt_state*, a stack that contains information necessary to backtrack to a previous state

3. *branch_iter*, which stores how far we are in exploring different possibilities from the current search node

4. *steps*, which remembers how many different unification bindings (such as imitation, projection, and identification) are applied

5. *subst*, a stack storing the variables bound so far

The iterator is initialized to hold the original problem in *constraints*, and all other fields are initially empty. The unifiers are retrieved one by one by calling the function ForwardIter. It returns True if the iterator made progress, in which case the unifier can be read via the iterator's *subst* field. Otherwise, no more unifiers can be found, and the iterator is no longer valid. The function's pseudocode is given below, including two auxiliary functions NormalizeHead and BacktrackIter:

```
function ForwardIter(iter)
  forward ← ¬iter.constraints.empty() ∨ BacktrackIter(iter)
  while forward ∧ ¬iter.constraints.empty() do
    (lhs, rhs) ← pop pair from iter.constraints
    lhs ← NormalizeHead(lhs)
    rhs ← NormalizeHead(rhs)
    normalize and discard the λ prefixes of lhs and rhs
    if ¬lhs.head.is_var() ∧ rhs.head.is_var() then
      swap lhs and rhs
    if lhs.head.is_var() then
      oracle_res ← Fixpoint(lhs, rhs, iter.subst)
      if oracle_res = NotInFragment then
        oracle_res ← Pattern(lhs, rhs, iter.subst)
      if oracle_res = NotUnifiable then
        forward ← BacktrackIter(iter)
      else if oracle_res = NotInFragment then
        n_steps, n_branch_iter, n_binding ←
          NextBinding(lhs, rhs, iter.steps, iter.branch_iter)
        if n_branch_iter ≠ BindEnd then
          push pair (lhs, rhs) back to iter.constraints
          push quadruple (iter.constraints, n_branch_iter,
            iter.steps, iter.subst) onto iter.bt_state
          extend iter.subst with n_binding
          iter.steps ← n_steps
          iter.branch_iter ← BindBegin
        else if lhs.head = rhs.head then
          push the constraint pairs of arguments of lhs and rhs to iter.constraints
          iter.branch_iter ← BindBegin
    else if lhs.head = rhs.head then
      push the constraint pairs of arguments of lhs and rhs to iter.constraints
    else
      forward ← BacktrackIter(iter)
  return forward
```

```
function NormalizeHead(t)
  if t.head = @ ∧ t.args[0].is_lambda() then
    reduce the top-level β-redex in t
    return NormalizeHead(t)
  else if t.head.is_var() ∧ t.head.binding ≠ nil then
    t.head ← t.head.binding
    return NormalizeHead(t)
  else
    return t
function BacktrackIter(iter)
  if iter.bt_state.empty() then
    clear all fields in iter
    return False
  else
    pop (constraints, branch_iter, steps, subst) from iter.bt_state
    set the corresponding fields of iter
    return True
```

ForwardIter begins by backtracking if the previous attempt was successful (i.e., all constraints were solved). If it finds a state from which it can continue, it takes term pairs from *constraints* until there are no more constraints or it is determined that no unifier exists. The terms are normalized by instantiating the head variable with its binding and then reducing the potential top-level β-redex that appears. This instantiation and reduction process is repeated until there are no more top-level β-redexes and the head is not a variable bound to some term. Then the term with shorter λ prefix is expanded (only on the top level) so that λ prefixes have the same length. Finally, the λ prefix is ignored, and we focus only on the body. In this way, we avoid fully substituting and normalizing terms and perform just enough operations to determine the next step of the procedure.

If either term of the constraint is flex, we first invoke oracles to solve the constraint. λE implements the most efficient oracles implemented in Zipperposition: fixpoint and pattern (Sect. 4.7). An oracle can return three results: (1) there is an MGU for the pair (Unifiable), which is recorded in *subst*, and the next pair in *constraints* is tried; (2) no MGU exists for the pair (NotUnifiable), which causes the iterator to backtrack; (3) if the pairs do not belong to the subclass that oracle can solve (NotInFragment), we generate possible variable bindings—that is, we guess the approximate form of the solution.

λE has a special module that generates bindings (NextBinding). This module is given the current constraint and the values of *branch_iter* and *steps*, and it either returns the next binding and the new values of *branch_iter* and *steps* or reports that all different variable bindings are exhausted. The bindings that λE's unification procedure creates are imitation, Huet-style projection, identification, and elimination (one argument at a time) (Sect. 4.3). A limit on the total number of applied binding rules can be set, as well as a limit on the number of individual rule applications. The binding module checks whether limits are reached using the iterator's *steps* field.

Computing bindings is the only point in the procedure where the search tree branches and different possibilities are explored. Thus, when λE follows the branch indicated by the binding module, it records the state to which it needs to return, should the followed branch

be backtracked. The state consists of the values of *constraints*, *steps*, and *subst* before the branch is followed and the value of *branch_iter* that points past the followed branch. The values of *branch_iter* are either BindBegin, which denotes that no binding was created, intermediate values that NextBinding uses to remember how far through bindings it is, and BindEnd, which indicates that all bindings are exhausted.

If all bindings are exhausted, the procedure checks whether the pair is flex–flex and both sides have the same head. If so, the pair is decomposed and constraints are derived from the arguments of the pair. Otherwise, the iterator backtracks. If the constraint is rigid–rigid, for unification to succeed, the heads of both sides of the constraint must be the same. Unification then continues with new constraints derived from the arguments. Otherwise, the iterator must be backtracked.

7.3.2 Matching

In E, the matching algorithm is mostly used inside simplification rules such as demodulation and subsumption [143]. As these rules must be efficiently performed, using any complex matching algorithm is not a viable option. Instead, we implemented a matching algorithm for the pattern class of terms [124] to complement Ehoh's λ-free higher-order matching algorithm (Sect. 3.4). A term is a *pattern* if each of its free variables either has no arguments (as in first-order logic) or is applied to distinct De Bruijn indices.

To determine which of the two algorithms to call (pattern or λ-free), we added a cached property `HasNonPatternVar`, which is set for terms of the form $X\,\overline{s}_n$ where $n > 0$ and either there exists some s_i that is not a De Bruijn index or there exist indices $i < j$ such that $s_i = s_j$ is a De Bruijn index. This property is propagated to the superterms when they are perfectly shared. This allows later checks if a term belongs to the pattern class to be performed in constant time.

We have modified the λ-free higher-order matching algorithm to treat λ prefixes as above in the unification procedure—by bringing the prefixes to the same length and ignoring them afterwards. This ensures that the algorithm will never try to match a free variable with a λ-abstraction, making sure that β-redexes never appear (as in original λ-free higher-order matching). We also modified the algorithm to ensure that free variables are never bound to terms that have loose bound variables. This algorithm cannot find many complex matching substitutions (matchers), but it can efficiently determine whether two terms are variable renamings of each other or whether a simple matcher can be used, as in the case of $(X\,(\lambda x.\,x)\,\mathtt{b}, \mathtt{f}\,(\lambda x.\,x)\,\mathtt{b})$, where $X \mapsto \mathtt{f}$ is usually the desired matcher. If this algorithm does not find a matcher and both terms are patterns, pattern matching is tried.

7.3.3 Indexing

E, like other modern theorem provers, efficiently retrieves unifiable or matchable pairs of terms using indexing data structures. To find terms unifiable with a query term or instances of a query term, it uses *fingerprint indexing* [144]. We extended this data structure to support (full) higher-order terms in Zipperposition (Sect. 4.6). We used the same approach in λE, and we extended feature vector indices [145] in the same way.

E uses *perfect discrimination trees* [113] to find generalizations of the query term (i.e., terms of which the query term is an instance). This data structure is a trie that indexes terms by representing them in a serialized, flattened form. The left branch from the root

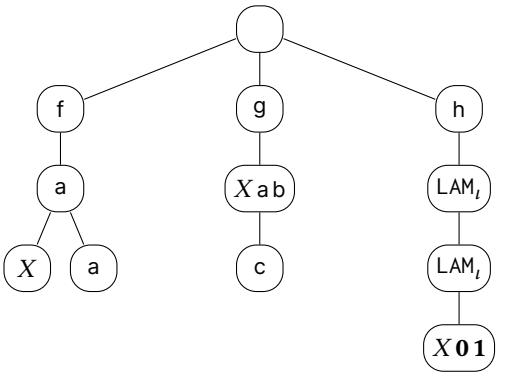


Figure 7.1: First-order, λ-free higher-order, and higher-order pattern terms in a perfect discrimination tree

in Figure 7.1 shows how the first-order terms $\mathtt{f}\,\mathtt{a}\,X$ and $\mathtt{f}\,\mathtt{a}\,\mathtt{a}$ are stored. In Ehoh, this data structure is extended to support partial application and applied variables (Sect. 3.5).

In λE, we extended this structure to support λ-abstractions and the higher-order pattern matching algorithm. To this end, we changed the way in which terms are serialized. First, we require that all terms are fully η-expanded (except for arguments of variables applied in patterns). Then, when the term is serialized, we dedicate a separate node for applied variable terms $X\,\overline{s}_n$, instead of the node for X followed by nodes for serialization of arguments $\overline{s}_n$. We serialize λ-abstraction $\lambda x.\, s$ using a special node $\mathtt{LAM}_\tau$, where τ is the type of x, followed by the serialization of s. Other than these two changes, serialization remains as in Ehoh, following the gracefulness principle. Figure 7.1 shows how $\mathtt{g}\,(X\,\mathtt{a}\,\mathtt{b})\,\mathtt{c}$ and $\mathtt{h}\,(\lambda x.\,\lambda y.\,X\,y\,x)$ are serialized.

Since the terms are stored in serialized form, it is hard to manipulate λ prefixes of stored terms during matching. Performing η-expansion when serializing terms makes sure that matchable terms have λ prefixes of the same length.

We have dedicated separate nodes for applied variables because access to arguments of applied variables is necessary for the pattern matching algorithm. Even though arguments can be obtained by querying the arity n of the variable and taking the next n arguments in the serialization, this is both inefficient and inelegant. As for De Bruijn indices, we treat them the same as function symbols and assign them their own nodes.

Following the notation from the extension of perfect discrimination trees to λ-free higher-order logic (Sect. 3.5), we now describe how enumeration of generalizations is performed. To traverse the tree, λE begins at the root node and maintains two stacks: `term_stack` and `term_proc`, where `term_stack` contains the subterms of the query term that have to be matched, and `term_proc` contains processed terms that are used to backtrack to previous states. Initially, `term_stack` contains the query term, the current matching substitution σ is empty, and the successor node is chosen among the child nodes as follows:

A. If the node is labeled with a symbol ξ (where ξ is either a De Bruijn index or a constant) and the top item t of `term_stack` is of the form $\xi\,\overline{t}_n$, replace t by n new items $t_1, \ldots, t_n$, and push t onto `term_proc`.

B. If the node is labeled with a symbol $\mathtt{LAM}_\tau$ and the top item t of `term_stack` is of the form $\lambda x.\, s$ and the type of x is τ, replace t by s, and push t onto `term_proc`.

C. If the node is labeled with a possibly applied variable $X\,\overline{s}_n$ (where $n \geq 0$), and the top item of `term_stack` is t, the matching algorithm described above is run on $X\,\overline{s}_n$ and t. The algorithm takes into account σ built so far and extends it if necessary. If the algorithm succeeds, pop t from `term_stack`, push it onto `term_proc`, and save the original value of σ in the node.

Backtracking works in the opposite direction: If the current node is labeled with a De Bruijn index or function symbol node of arity n, pop n terms from `term_stack` and move the top of `term_proc` to `term_stack`. If the node is labeled with $\mathtt{LAM}_\tau$, pop the top of `term_stack` and move the top of `term_proc` to `term_stack`. Finally, if the node is labeled with a possibly applied variable, move the top of the `term_proc` to `term_stack` and restore the value of σ.

As an example of how finding a generalization works, consider the following states of stacks and substitutions, which emerge when looking for generalizations of `g(fab)c` in the tree of Figure 7.1:

	ϵ	g	g.(X a b)	g.(X a b).c
σ:	$\emptyset$	$\emptyset$	$\{X \mapsto \mathsf{f}\}$	$\{X \mapsto \mathsf{f}\}$
`term_stack`:	[g(fab)c]	[fab, c]	[c]	[]
`term_proc`:	[]	[g(fab)c]	[fab, g(fab)c]	[c, fab, g(fab)c]

7.4 Preprocessing, Calculus, and Extensions

Ehoh's simple λ-free higher-order calculus performed well on Sledgehammer problems and formed a promising stepping stone to full higher-order logic (Sect. 3.9). When implementing support for full higher-order logic, we were guided by efficiency and gracefulness with respect to Ehoh's calculus rather than completeness. Whereas Zipperposition provides both complete and incomplete modes, λE only offers incomplete modes.

Preprocessing Our experience with Zipperposition showed the importance of flexibility in preprocessing the higher-order problems (Sect. 6.3). Therefore, we implemented a flexible preprocessing module in λE.

To maintain compatibility with Ehoh, λE can optionally transform all λ-abstractions into named functions. This process is called λ-*lifting* [83]. λE also removes all occurrences of Boolean subterms (other than $\mathsf{T}, \bot$, and free variables) in higher-order contexts using a FOOL-like transformation [98]. For example, the formula $\mathsf{f}(\mathsf{p} \wedge \mathsf{q}) \approx \mathsf{a}$ becomes $(\mathsf{p} \wedge \mathsf{q} \longrightarrow \mathsf{f}(\mathsf{T}) \approx \mathsf{a}) \wedge (\neg(\mathsf{p} \wedge \mathsf{q}) \longrightarrow \mathsf{f}(\bot) \approx \mathsf{a})$.

Many TPTP problems use the `definition` role to denote the definitions of symbols. λE can treat the definition axioms as rewrite rules, and replace all occurrences of defined symbols during preprocessing. Furthermore, during SInE [80] axiom selection, it can always include the defined symbol in the trigger relation.

Calculus λE implements the same superposition calculus as Ehoh with three important changes. First, wherever Ehoh requires the MGU of terms, λE enumerates unifiers from a finite subset of the CSU, as explained in Sect. 7.3. Second, λE uses versions of the KBO and LPO orders designed for λ-terms.

The third difference is more subtle. One of the main features of Ehoh is *prefix optimization* (Sect. 3.1): a method that, given a demodulator $s \approx t$, allows to replace both applied and unapplied occurrences of s by t by traversing only first-order subterms of a rewritable term. In a λ-free setting this optimization is useful, but in the presence of $\beta\eta$-normalization, shapes of terms can change drastically, making it much harder to track prefixes of terms. This is why we disabled the prefix optimization in λE. To counterbalance this removal, we introduced the argument congruence rule AC in λE and enabled positive and negative functional extensionality (PE and NE) by default:

$$\frac{s \approx t \vee C}{s\,X \approx t\,X \vee C}\,\mathrm{AC} \qquad \frac{s \not\approx t \vee C}{s\,(\mathsf{sk}\,\overline{X}) \not\approx t\,(\mathsf{sk}\,\overline{X}) \vee C}\,\mathrm{NE} \qquad \frac{s\,X \approx t\,X \vee C}{s \approx t \vee C}\,\mathrm{PE}$$

AC and NE assume that s and t are of function type. In NE, $\overline{X}$ denotes all the free variables occurring in s and t, and sk is a fresh Skolem symbol of the appropriate type. PE has a side condition that X may not appear in s, t, or C.

Saturation E's saturation procedure is based on the assumption that each attempt to perform an inference will either result in a single clause or fail due to one of the inference side conditions. Unification procedures that produce multiple substitutions break this invariant, and the saturation procedure needed to be adjusted.

In Sect. 6.6, we described a variant of the saturation procedure that elegantly supports interleaving computations of unifiers and scheduling inferences to be performed. Since completeness was not one of the λE design goals, we did not implement this version of the saturation procedure. Instead, in places where previously a single unifier was expected, λE consumes all elements of the iterator used for enumerating a unifier, converting them to clauses.

Reasoning about Formulas Even though most of the Boolean structure is removed during preprocessing, formulas can reappear at the top level of clauses during saturation. For example, after replacing X with $\lambda x.\,\lambda y.\,x \wedge y$, the clause $X\,\mathsf{p}\,\mathsf{q} \vee \mathsf{a} \approx \mathsf{b}$ becomes $(\mathsf{p} \wedge \mathsf{q}) \vee \mathsf{a} \approx \mathsf{b}$. λE converts every clause of the form $\varphi \vee C$, where φ has a logic symbol as its head (or it is a (dis)equation between two formulas different than T), to an explicitly quantified formula. Then, the clausification algorithm is invoked on the formula to restore the clausal structure. Zipperposition features more dynamic clausification modes, but for simplicity we decided not to implement them in λE.

The *o*λSup calculus [17] includes many rules that act on Boolean subterms, which are necessary for completeness. Other than Boolean simplification rules, which use simple tautologies such as $\mathsf{p} \wedge \mathsf{T} \leftrightarrow \mathsf{p}$ to simplify terms, we have implemented none of the Boolean rules of this calculus in λE. First, we have observed that complicated rules such as FluidBoolHoist and FluidLoobHoist are hardly ever useful in practice and usually only contribute to an uncontrolled increase in the proof state size. Second, simpler rules

such as BoolHoist can usually be simulated by pragmatic rules that perform Boolean and functional extensionality reasoning, described below.

To make up for excluding Boolean rules, we use an incomplete, but more easily controllable and intuitive rule, called *primitive instantiation*. This rule instantiates free predicate variables with approximations of formulas that are ground instances of this variable. We described this rule in Sect. 5.3. In λE it is implemented in a similar manner.

λE's handling of the Hilbert choice operator is inspired by Leo-III's [153]. λE recognizes clauses of the form $\neg P\, X \vee P(\mathsf{f}\, P)$ which essentially denote that f is a choice symbol. Then, when subterm $\mathsf{f}\, s$ is found during saturation, s is used to instantiate the choice axiom for f. Similarly, Leibniz equality is eliminated by recognizing clauses of the form $\neg P\, \mathsf{a} \vee P\, \mathsf{b} \vee C$. These clauses are then instantiated with $P \mapsto \lambda x.\, x \approx \mathsf{a}$ and $P \mapsto \lambda x.\, x \not\approx \mathsf{b}$, which results in $\mathsf{a} \approx \mathsf{b} \vee C$. Both rules are described in Sect. 5.3 in more detail.

Finally, λE treats induction axioms specially. Just like Zipperposition (Sect. 6.4), it abstracts literals from the goal clauses and uses these abstractions to instantiate induction axioms. Since Zipperposition supports dynamic calculus-level clausification, instantiation of induction axioms happens during saturation, when these axioms are processed. In λE, this instantiation is performed statically, immediately after clausification. After λE collects all the abstractions, it traverses the clauses and instantiates those that have applied variable of the same type as the abstraction.

Extensionality λE takes a pragmatic approach to reasoning about functional and Boolean extensionality: It uses *abstracting* rules (Abs rules of Sect. 5.3) that simulate basic superposition calculus rules, but do not require unifiability of the partner terms in the inference. More precisely, assume a core inference needs to be performed between two β-reduced terms u and v, such that they can be represented as $u = C[s_1, \ldots, s_n]$ and $v = C[t_1, \ldots, t_n]$, where C is the most general common context (more precisely, green common context that does not go into the structure of λ-abstraction or applied variables [18]) of u and v, not all of s_i and t_j are free variables, and for at least one i, $s_i \neq t_i$, s_i and t_i are not (possibly applied) free variables, and they are of Boolean or function type. Then, the conclusion is formed by taking the conclusion D of the core inference rule (which would be created if s and t are unifiable) and adding literals $s_1 \not\approx t_1 \vee \cdots \vee s_n \not\approx t_n$.

These rules are particularly useful because λE has no rules that dynamically process Booleans in FOOL-like fashion, such as BoolHoist. For example, given the clauses $\mathsf{f}(\mathsf{p} \wedge \mathsf{q}) \approx \mathsf{a}$ and $\mathsf{g}(\mathsf{f}\, \mathsf{p}) \not\approx \mathsf{b}$, the abstracting version of the superposition rules SP and SN (Sect. 2.5.3) would result in $\mathsf{g}\, \mathsf{a} \not\approx \mathsf{b} \vee (\mathsf{p} \wedge \mathsf{q}) \not\approx \mathsf{p}$. In this way, the Boolean structure bubbles up to the top level and is further processed by clausification. We noticed that this alleviates the need for the other Boolean rules in practice.

7.5 Evaluation

In this section, we try to answer two questions about λE: *How does λE compare against other higher-order provers (including Ehoh)? Does λE introduce any overhead compared with Ehoh?* To answer these questions, we ran provers on problems from the TPTP library [157] and on benchmarks generated by Sledgehammer (SH) [131]. The experiments were carried out on StarExec Miami [154] nodes equipped with Intel Xeon E5-2620 v4 CPU clocked at

2.10 GHz. For the TPTP part, we used the CASC-28[1] time limits: 120 s wall-clock and 960 s CPU. For SH benchmarks and to answer the other question, we used Sledgehammer's default time limit: 30 s wall-clock and CPU. The raw evaluation data is available online[2].

Comparison with Other Provers To answer the first question, we let λE compete with the top contenders in the higher-order division of CASC-28: cvc5 0.0.7,[3] Ehoh 2.7, Leo-III 1.6.6 [153], Vampire 4.6 [25], and Zipperposition 2.1 (Chapter 6). We also included Satallax 3.5 [39]. We used all 2899 higher-order theorems in TPTP 7.5.0 as well as 5000 SH higher-order benchmarks originating from the Seventeen benchmark suite [55]. On SH benchmarks, cvc5, Ehoh, λE, Vampire, and Zipperposition were run using custom schedules provided by their developers, optimized for single-core usage and low timeouts. Otherwise, we used the corresponding CASC configurations. Note that Ehoh 2.7 is an updated version of Ehoh described in Chapter 3, which can parse not only λ-free but also full higher-order. It is only a syntactic extension as λ-abstractions are supported using λ-lifting [83]. We included two versions of Zipperposition: *coop* uses Ehoh 2.7 as a backend to finish proof attempts, whereas *uncoop* does not use this feature. The results are shown in Figure 7.2.

Both Ehoh and λE were run in the automatic scheduling mode. Compared to Ehoh, λE features a redesigned module for automatic scheduling, it can use multiple CPU cores, and its heuristics have been trained better on higher-order problems.

λE dramatically improves the higher-order reasoning capabilities compared to Ehoh. It solves 20% more problems on TPTP benchmarks and 7% more problems on SH benchmarks, where Ehoh was already very successful.

λE was mainly designed as an efficient backend to proof assistants. As such it excels on SH benchmarks, outperforming the competition. On TPTP, it outperforms all higher-order provers other than Zipperposition-coop. If Zipperposition's Ehoh backend is disabled, λE outperforms Zipperposition by a wide margin. This comparison is arguably fairer; after all, λE does not use an older version of Zipperposition as a backend. These results suggest that λE already implements most of the necessary features for a high-performance higher-order prover but could benefit from the kind of fine-tuning that Zipperposition underwent in the last two years.

Remarkably, there is a substantial disparity in the set of problems solved by λE and Zipperposition-coop. The raw data show thats λE solves 181 SH problems and 24 TPTP problems that Zipperposition-coop does not. The lower number of uniquely solved TPTP problems is most likely due to Zipperposition's being heavily optimized on TPTP.

Comparison with the First-Order E Both Ehoh and λE can be compiled in a mode that disables most of the higher-order reasoning. This mode is designed for users that are interested only in E's first-order capabilities and care a lot about performance. To answer the second evaluation question, about assessing overhead of λE, we chose all the 1138 unique problems used at CASC from 2019 to 2021 in the first-order theorem division and ran Ehoh and λE both in this first-order (FO) mode and in higher-order (HO) mode using 30 s CPU and wallclock timeout.

[1] http://www.tptp.org/CASC/28/
[2] https://doi.org/10.5281/zenodo.6389849
[3] https://cvc5.github.io/

	TPTP	SH
cvc5	1931	2577
Ehoh	2105	2611
λE	2533	**2804**
Leo-III	2282	1601
Satallax	2320	1719
Vampire	2203	2240
Zipperposition-coop	**2583**	2754
Zipperposition-uncoop	2483	2181

Figure 7.2: Comparison of higher-order provers

	TPTP
Ehoh FO	535
Ehoh HO	538
λE FO	537
λE HO	**541**

Figure 7.3: Evaluation of λE's overhead

We fixed a single configuration of options, because Ehoh's and λE's automatic scheduling methods could select different configurations and we would not be measuring the overhead but the quality of the chosen configurations. We chose the *boa* configuration (Sect. 3.7), which is the configuration that E 2.2 used most often in its automatic scheduling mode. The results are shown in Figure 7.3.

Counterintuitively, the higher-order versions of both provers outperform the first-order counterparts. However, the difference is so small that it can be attributed to the changes to memory layout that affect the order in which clauses are chosen. Similar effects are seen when comparing the first-order versions. We would expected the less heavily modified version of E, Ehoh, to perform better, but due to subtle effects λE wins.

7.6 Discussion and Related Work

On the trajectory to λE, we developed, together with colleagues, three superposition calculi: λfSup for λ-free higher-order logic [15], λSup for a higher-order logic with λ-abstraction but no Booleans [18], and *o*λSup for full higher-order logic [17]. These milestones allowed us to carefully estimate how the increased reasoning capabilities of each calculus influence its performance. We also developed a complete and efficient unification algorithm (Chapter 4) which can easily be customized to trade bits of its completeness for efficiency. Practically oriented work, described in Chapters 5 and 6, helped us gauge how much reasoning power we lose by using more intuitive and more easily controllable incomplete rules. Both the theoretical and the practical lines of our work met in this chapter, resulting in a prover that outperforms the competition on the benchmarks from proof assistants.

Extending first-order provers with higher-order reasoning capabilities has been attempted by other researchers as well. Barbosa et al. extended the SMT solvers CVC4 (now cvc5) and veriT to higher-order logic in an incomplete way [11]. Bhayat and Reger first extended Vampire to higher-order logic using combinatory unification [24], an incomplete approach, before they designed and implemented a complete higher-order superposition calculus based on SKBCI-combinators [25]. The advantage is that combinators can be supported as a thin layer on top of λ-free terms. This calculus is also implemented in Zipperposition. However, in informal experiments, we found that λ-superposition per-

forms substantially better, corroborating the CASC results, so we decided to make a more profound change to Ehoh and implement λ-superposition.

Possibly the only actively maintained higher-order prover built from the bottom up as a higher-order prover is Leo-III [153], as the long-time winner of CASC higher-order division, Satallax [39], is no longer maintained. A further overview of other traditional higher-order provers and the calculi they are based on can be found in Sect. 3.10.

7.7 Conclusion

In 2019, the reviewers of our paper introducing Ehoh [169] were skeptical that extending it with support for full higher-order logic would be feasible. One of them wrote:

> A potential criticism could be that this step from E to Ehoh is just extending FOL by those aspects of HOL that are easily in reach with rather straightforward extensions (none of the extensions is indeed very complicated), and that the difficult challenges of fully supporting HOL have yet to be confronted.

We ended up addressing the theoretical "difficult challenges" in other work with colleagues. In this chapter, we faced the practical challenges pertaining to the extension of Ehoh's data structures and algorithms to support full higher-order logic and demonstrated that such an extension is possible. Our evaluation shows that this extension makes λE the best higher-order prover on benchmarks coming from interactive theorem proving practice, which was our goal. λE lags slightly behind Zipperposition on TPTP problems, possibly because Zipperposition does not assume a clausal structure and can perform subtle formula-level inferences. In the future, we plan to implement the same features in λE. We have also only started tuning λE's heuristics on higher-order problems.

8

SAT-Inspired Eliminations for Superposition

Joint work with
Jasmin Blanchette and Marijn J.H. Heule

Optimized SAT solvers simplify the clause set not only during preprocessing, but they also simplify it during solving. This interleaving of simplification and solving is called inprocessing. Some preprocessing techniques have been generalized to first-order logic with equality. In this chapter, we port inprocessing techniques to work with superposition, a leading first-order proof calculus, and strengthen known preprocessing techniques. Specifically, we look into elimination of hidden literals, variables (predicates), and blocked clauses. Our evaluation using the Zipperposition prover confirms that the new techniques usefully supplement the existing superposition machinery.

In this work I was the main designer of all the presented techniques. Marijn Heule did weekly supervision and provided his knowledge of SAT solving to guide the design of all techniques. Jasmin Blanchette found the exact conditions under which superposition remains complete when the predicate elimination rule is added to the calculus. I also implemented and evaluated all the techniques.

8.1 Introduction

Automated reasoning tools have become much more powerful in the last few decades, thanks to procedures such as conflict-driven clause learning (CDCL) [112] for propositional logic and superposition for first-order logic with equality. However, the effectiveness of these procedures crucially depends on how the input problem is represented as a clause set. The clause set can be optimized beforehand (*preprocessing*) or during the execution of the procedure (*inprocessing*). In this chapter, we lift several preprocessing and inprocessing techniques from propositional logic to clausal first-order logic, and demonstrate their usefulness in a superposition prover.

For many years, SAT solvers have used inexpensive clause simplification techniques such as *hidden literal* and *hidden tautology elimination* [76, 77] and *failed literal detection* [66, Sect. 1.6]. We generalize these techniques to first-order logic with equality (Sect. 8.3). Since the generalization involves reasoning about infinite sets of literals, we propose restrictions to make them usable.

Variable elimination, based on Davis–Putnam resolution [52], has been studied in the context of both propositional logic [46, 155] and quantified Boolean formulas (QBFs) [26]. The basic idea is to resolve all clauses with negative occurrences of a propositional variable (i.e., a nullary predicate symbol) against clauses with positive occurrences and delete the parent clauses. Eén and Biere [59] refined the technique to identify a subset of clauses that effectively define a variable and use it to further optimize the clause set. This latter technique, *variable elimination by substitution*, has been an important preprocessor component in many SAT solvers since its introduction in 2004.

Specializing second-order quantifier elimination [67, 128], Khasidashvili and Korovin [91] adapted variable elimination to preprocess first-order problems, yielding a technique we call *singular predicate elimination*. We extend their work along two axes (Sect. 8.4): We generalize Eén and Biere's refinement to first-order logic, resulting in *defined predicate elimination*, and explain how both types of predicate elimination can be used during the proof search as inprocessing.

The last technique we study is *blocked clause elimination* (Sect. 8.5). It is used in both SAT [85] and QBF solvers [27]. Its generalization to first-order logic has produced good results when used as a preprocessor, especially on satisfiable problems [93]. We explore more ways to use blocked clause elimination on satisfiable problems, including using it to establish equisatisfiability with the empty clause set or as an inprocessing rule. Unfortunately, we find that its use as inprocessing can compromise the refutational completeness of superposition.

All techniques have been implemented in the Zipperposition prover (Sect. 8.6), allowing us to ascertain their usefulness (Sect. 8.7). The best configuration solves 160 additional problems on benchmarks consisting of all 13 495 first-order TPTP theorems [157]. The raw experimental data are publicly available.[1]

8.2 Preliminaries

Our setting is many-sorted (i.e., many-typed), first-order logic [68] with interpreted equality and a distinguished type (or sort) o. We introduced this logic in Sect. 2.2 and here we

[1] https://doi.org/10.5281/zenodo.4552499

use the same notation introduced in that section. We write $\mathsf{f}^i(s)$ for the i-fold application of an unary symbol f (e.g., $\mathsf{f}^3(x) = \mathsf{f}(\mathsf{f}(\mathsf{f}(x)))$) and call nullary function symbols *constants*. In this chapter, propositional variables (corresponding to predicate constants in first-order logic) are written as $\mathsf{x}, \mathsf{y}, \mathsf{p}, \mathsf{q}, \ldots$, while first-order free variables are written in lowercase as $x, y, z, \ldots$.

We assume the notion of substitution introduced in Sect. 2.2. We extend the notation by writing a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ shortly as $\{\overline{x} \mapsto \overline{t}\}$. A substitution is a *variable renaming* if it is a bijection from a set of variables to a set of variables. Iterated, i-fold application of a nonidempotent application σ is denoted by σ^i.

As we are working with clausal logic, we assume the clausal structure as defined in Sect. 2.4. In this section we use the uppercase letter L to denote literals, as this is more customary in the related literature for propositional logic. Recall that predicate literals are encoded as (dis)equations with equality. To simplify the notation in this chapter we write predicate literals in unencoded form: Positive literal $s \approx \mathsf{T}$ is written as s and negative literal $s \not\approx \top$ is written as $\neg s$. Literals $s \approx t$ where neither s nor t have the type o are called *functional*. Given a literal L, we overload notation and write $\neg L$ to denote its complement. Clauses are often defined as sets of literals, but superposition needs multisets; with multisets, an instance $\sigma(C)$ always has the same number of literals as C, a most convenient property. Given a clause set N, $N\downarrow_2$ denotes the subset of its binary clauses: $N\downarrow_2 = \{L_1 \vee L_2 \mid L_1 \vee L_2 \in N\}$.

We assume the natural extensions of domain, valuation, interpretation, and model (as defined by Fitting [65]) from unsorted to many-sorted logic. We also assume the notion of normal models from Sect. 2.2 (with corresponding notation) and extend it with the notion of *canonical models*: A canonical model $\mathcal{J}$ is a normal model such that for every element d in one of $\mathcal{J}$'s domains there exists a ground term t such that $\mathcal{J}$ interprets t as d. We also make use of Herbrand models and Herbrand's theorem [65, Sect. 5.4]. In Herbrand models the domain consists of all ground terms. These models have a useful property that a formula is satisfiable if and only if it is satisfiable in an Herbrand model. Herbrand's theorem states that every unsatisfiable clause set has (propositionally) unsatisfiable ground instance. Canonical models generalize Herbrand models to first-order logic with equality.

All of the techniques described in this chapter are applied in the context of first-order superposition which was introduced in Sect. 2.5. We reuse all the notions introduced in this section and the corresponding notations.

8.3 Hidden-Literal-Based Elimination

In propositional logic, binary clauses from a clause set N can be used to efficiently discover literals L, L' for which the implication $L' \longrightarrow L$ is entailed by N's binary clauses—i.e., $N\downarrow_2 \models L' \longrightarrow L$. Heule et al. [77] introduced the concept of *hidden literals* to capture such implications.

Definition 8.1. Given a propositional literal L and a propositional clause set N, the set of *propositional hidden literals* for L and N is $\mathrm{HL}_\mathrm{p}(L, N) = \{L' \mid L' \hookrightarrow_\mathrm{p}^* L\} \setminus \{L\}$, where $\hookrightarrow_\mathrm{p}$ is defined so that $\neg L_1 \hookrightarrow_\mathrm{p} L_2$ whenever $L_1 \vee L_2 \in N$. Moreover, $\mathrm{HL}_\mathrm{p}(L_1 \vee \cdots \vee L_n, N) = \bigcup_{i=1}^n \mathrm{HL}_\mathrm{p}(L_i, N)$.

Heule et al. used a definition based on a fixpoint computation, but our definition based on the reflexive transitive closure $\hookrightarrow_{\mathrm{p}}^{*}$ of $\hookrightarrow_{\mathrm{p}}$ is equivalent. Intuitively, a hidden literal can be added to or removed from a clause without affecting its semantics in models of N. By eliminating hidden literals from C, we simplify it. By adding hidden literals to C, we might get a tautology C' (i.e., a valid clause: $\models C'$), meaning that $N{\downarrow_2} \models C$, thereby enabling us to delete C. Note that $\mathrm{HL}_{\mathrm{p}}(L, N)$ is finite for a finite N.

Definition 8.2. Given $L' \vee L \vee C \in N$, if $L' \in \mathrm{HL}_{\mathrm{p}}(L, N)$, *hidden literal elimination* (HLE) replaces N by $(N \setminus \{L' \vee L \vee C\}) \cup \{L \vee C\}$. Given $C \in N$, $\{L_1, \ldots, L_n\} = \mathrm{HL}_{\mathrm{p}}(C, N)$, and $C' = C \vee L_1 \vee \cdots \vee L_n$, if C' is a tautology, *hidden tautology elimination* (HTE) replaces N by $N \setminus \{C\}$.

Theorem 8.3. *The result of applying HLE or HTE to a clause set N is equivalent to N.*

Proof. For HLE, if $L' \in \mathrm{HL}_{\mathrm{p}}(L, N)$, $N{\downarrow_2} \models \neg L' \vee L$. Then, subsumption resolution (a sound rule that applies resolution followed by subsumption [8]) yields shortened clause $L \vee C'$ from Definition 8.2. For HTE, it can be shown that sets N and $N \cup \{C'\} \setminus \{C\}$ are equivalent [77, Sect. 2.1]. As clause C' is a tautology, N and $N \setminus \{C\}$ are equivalent. □

We generalize hidden literals to first-order logic with equality by considering substitutivity of variables as well as congruence of equality.

Definition 8.4. Given a literal L and a clause set N, the set of *hidden literals* for L and N is $\mathrm{HL}(L, N) = \{L' \mid L' \hookrightarrow^{*} L\} \setminus \{L\}$, where $\hookrightarrow$ is defined so that:

1. $\sigma(\neg L') \hookrightarrow \sigma(L)$ if $L' \vee L \in N$ and σ is a substitution
2. $s \approx t \hookrightarrow u[s] \approx u[t]$ for all terms s, t and contexts $u[\,]$
3. $u[s] \not\approx u[t] \hookrightarrow s \not\approx t$ for all terms s, t and contexts $u[\,]$

Moreover, $\mathrm{HL}(L_1 \vee \cdots \vee L_n, N) = \bigcup_{i=1}^{n} \mathrm{HL}(L_i, N)$.

The generalized definition also enjoys the key property that $L' \in \mathrm{HL}(L, N)$ implies $N{\downarrow_2} \models L' \longrightarrow L$. However, $\mathrm{HL}(L, N)$ may be infinite even for predicate literals; for example, $\mathsf{p}(\mathsf{f}^i(x)) \in \mathrm{HL}(\mathsf{p}(x), \{\mathsf{p}(x) \vee \neg\mathsf{p}(\mathsf{f}(x))\})$ for every i.

Based on Definition 8.4, we can generalize hidden literal elimination and support a related technique:

$$\frac{L' \vee L \vee C}{L \vee C}\,\mathrm{HLE} \quad \text{if } L' \in \mathrm{HL}(L, N)$$

$$\frac{L \vee C}{C}\,\mathrm{FLE} \quad \text{if } L', \neg L' \in \mathrm{HL}(\neg L, N)$$

Recall that double lines denote simplification rules (Sect. 2.5). The second rule is called *failed literal elimination*, inspired by the SAT technique of asserting $\neg L$ if L is a *failed literal* [66]. It is easy to see that rule HLE is sound. From $L' \in \mathrm{HL}(L, N)$ we have $N \models L' \longrightarrow L$ (i.e., $\neg L' \vee L$). Performing subsumption resolution [8] between $L' \vee L \vee C$ and $\neg L' \vee L$ yields the conclusion, which is therefore entailed by N. For FLE, the condition $L', \neg L' \in \mathrm{HL}(\neg L, N)$ means that $N{\downarrow_2} \models \{\neg L' \vee \neg L, L' \vee \neg L\} \models \neg L$.

Example 8.5. Consider the clause set $N = \{\mathsf{p}(x) \vee \neg\mathsf{p}(\mathsf{f}(x)), \mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \vee \mathsf{a} \approx \mathsf{b}\}$ and the clause $C = \mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \vee \mathsf{p}(x)$. The first clause in N induces $\mathsf{p}(\mathsf{f}(x)) \hookrightarrow \mathsf{p}(x)$, $\mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \hookrightarrow \mathsf{p}(\mathsf{f}(x))$, and hence $\mathsf{p}(\mathsf{f}(\mathsf{f}(x))) \hookrightarrow^* \mathsf{p}(x)$. Together with the second clause in N, it can be used to derive $\mathsf{a} \not\approx \mathsf{b} \hookrightarrow^* \mathsf{p}(x)$. Finally, using rule 3 of Definition 8.4, we derive $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \hookrightarrow^* \mathsf{p}(x)$—that is, $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(\mathsf{b}) \in \mathrm{HL}(\mathsf{p}(x), N)$. This allows us to remove C's first literal using HLE.

Two special cases of HLE exploit equality congruence as embodied by conditions 2 and 3 of Definition 8.4 without requiring to compute the HL set:

$$\frac{s \approx t \vee u[s] \approx u[t] \vee C}{u[s] \approx u[t] \vee C}\,\textsc{CongHLE}^+$$

$$\frac{s \not\approx t \vee u[s] \not\approx u[t] \vee C}{s \not\approx t \vee C}\,\textsc{CongHLE}^-$$

Hidden literals can be combined with unit clauses L' to remove more literals:

$$\frac{L' \quad L \vee C}{L' \quad C}\,\textsc{UnitHLE} \quad \text{if } \sigma(L') \in \mathrm{HL}(\neg L, N)$$

Given a unit clause $L' \in N$, the rule uses L' to discharge $\sigma(L')$ in $N \models \sigma(L') \longrightarrow \neg L$. As a result, we have $N \models \neg L$, making it possible to remove L from $L \vee C$.

Example 8.6. Consider the clause set $N = \{\mathsf{p}(x) \vee \mathsf{q}(\mathsf{f}(x)), \neg\mathsf{q}(\mathsf{f}(\mathsf{a})) \vee \mathsf{f}(\mathsf{b}) \approx \mathsf{g}(\mathsf{c}), \mathsf{f}(x) \not\approx \mathsf{g}(y)\}$ and the clause $C = \neg\mathsf{p}(\mathsf{a}) \vee \neg\mathsf{q}(\mathsf{b})$. The first clause in N induces $\neg\mathsf{q}(\mathsf{f}(\mathsf{a})) \hookrightarrow \mathsf{p}(\mathsf{a})$, whereas the second one induces $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{c}) \hookrightarrow \neg\mathsf{q}(\mathsf{f}(\mathsf{a}))$. Thus, we have $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{g}(\mathsf{c}) \hookrightarrow^* \mathsf{p}(\mathsf{a})$—that is, $\mathsf{f}(\mathsf{b}) \not\approx \mathsf{f}(\mathsf{c}) \in \mathrm{HL}(\mathsf{p}(\mathsf{a}), N)$. By applying the substitution $\{x \mapsto \mathsf{b},\, y \mapsto \mathsf{c}\}$ to the third clause in N, we can fulfill the conditions of UnitHLE and remove C's first literal.

Next, we generalize hidden tautologies to first-order logic.

Definition 8.7. A clause C is a *hidden tautology* for a clause set N if there exists a finite set $\{L_1, \ldots, L_n\} \subseteq \mathrm{HL}(C, N)$ such that $C \vee L_1 \vee \cdots \vee L_n$ is a tautology.

Example 8.8. In general, hidden tautologies are not redundant and cannot be deleted during saturation. Consider the unsatisfiable set $N = \{\neg\mathsf{a}, \neg\mathsf{b}, \mathsf{a} \vee \mathsf{c}, \mathsf{b} \vee \neg\mathsf{c}\}$, the order $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c}$, and the empty selection function. The only possible superposition inference from N is between the last two clauses, yielding the hidden tautology $\mathsf{a} \vee \mathsf{b}$ (after simplifying away $\top \not\approx \top$), which is entailed by the larger clauses $\mathsf{a} \vee \mathsf{c}$ and $\mathsf{b} \vee \neg\mathsf{c}$. If this clause is removed, the prover could enter an infinite loop, forever generating and deleting the hidden tautology and never getting the opportunity to derive the empty clause.

In practice, most provers use a variant of the given clause procedure. Removing hidden tautologies breaks the invariant of the procedure that all inferences between clauses in $\mathscr{A}$ are redundant. The end result is not that the prover diverges, but that it terminates without deriving the empty clause.

To observe this, assume the setting as in Example 8.8, and let $\mathscr{P} = N$ and $\mathscr{A} = \emptyset$. After moving the first three clauses from $\mathscr{P}$ to $\mathscr{A}$ ($\mathscr{A} = \{\neg\mathsf{a}, \neg\mathsf{b}, \mathsf{a} \vee \mathsf{c}\}$, $\mathscr{P} = \{\mathsf{b} \vee \neg\mathsf{c}\}$), no inferences

are possible, and no new clauses are added to $\mathscr{P}$. After the last clause is moved to $\mathscr{A}$, the hidden tautology $\mathtt{a} \vee \mathtt{b}$ is produced. If it is deleted, the prover terminates with the unsatisfiable set $\mathscr{A}$, but does not derive the empty clause.

To delete hidden tautologies during saturation, the prover could check that all the relevant clause instances encountered along the computation of HL are $\prec$-smaller than a given hidden tautology. However, this would be expensive and seldom succeed, given that superposition creates lots of nonredundant hidden tautologies. Instead, we propose to simplify hidden tautologies using the following rules:

$$\frac{L \vee L' \vee C}{L \vee L'}\,\text{HTR} \quad \text{if } \neg L' \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

$$\frac{L \vee C}{L}\,\text{FLR} \quad \text{if } L', \neg L' \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

We call these techniques *hidden tautology reduction* and *failed literal reduction*, respectively. Both rules are sound. As with hidden literals, unit clauses L' can be exploited:

$$\frac{L' \quad L \vee C}{L' \quad L}\,\textsc{UnitHTR} \quad \text{if } \sigma(L') \in \mathrm{HL}(L, N) \text{ and } C \neq \bot$$

We give the simplification rules above (for hidden literal elimination, hidden tautology reduction, failed literal detection, and their variants) the collective name of *hidden-literal-based elimination* (HLBE). Yet another use of hidden literals is for *equivalent literal substitution* [76]: If both $L' \in \mathrm{HL}(L, N)$ and $L \in \mathrm{HL}(L', N)$, we can often simplify $\sigma(L')$ to $\sigma(L)$ in N if $\sigma(L') \succ \sigma(L)$. We want to investigate this further.

Theorem 8.9. *The rules HLE, FLE, CongHLE$^+$, CongHLE$^-$, UnitHLE, HTR, FLR, and UnitHTR are sound simplification rules.*

Proof. It is easy to see that the deleted premises are entailed by the conclusions that replace them and that the conclusions' instances are $\prec$-smaller than the premises' instances, as required by the redundancy criterion. It remains to check soundness.

Case HLE: We have $N, L' \models L$ by the side condition and must show $N, L' \vee L \vee C \models L \vee C$. Let $\mathscr{J} \models N, L' \vee L \vee C$. If $\mathscr{J} \models L'$, then we also have $\mathscr{J} \models L$ thanks to the side condition and hence $\mathscr{J} \models L \vee C$. Otherwise, we have $\mathscr{J} \models L \vee C$, which is exactly what we need to show.

Case FLE: We have $N, L \models L'$ and $N, L \models \neg L'$ by the side condition. If $\mathscr{J} \models N, L$, then both $\mathscr{J} \models L'$ and $\mathscr{J} \models \neg L'$, an absurdity. Otherwise, we have $\mathscr{J} \models C$, as desired.

Case CongHLE$^+$, CongHLE$^-$: Obvious by congruence of equality.

Case UnitHLE: We have $N, L \models \neg\sigma(L')$ by the side condition. If $\mathscr{J} \models N, L$, then $N \models \neg\sigma(L')$. But since $L' \in N$, this is an absurdity. Otherwise, we have $\mathscr{J} \models C$, as desired.

Case HTR: We have $N, \neg L' \models L$ by the side condition. If either $\mathscr{J} \models L$ or $\mathscr{J} \models L'$, the desired result follows directly. Otherwise, from $\mathscr{J} \models \neg L'$ we also have $\mathscr{J} \models L$ thanks to the side condition, contradicting $\mathscr{J} \models \neg L$.

Case FLR: We have $N, L' \models L$ and $N, \neg L' \models L$ by the side condition. Hence $N \models L$, as desired.

Case UnitHTR: We have $N, \sigma(L') \models L$. Since $L' \in N$, we have $N \models L$, as desired. □

8.4 Predicate Elimination

For propositional logic, variable elimination [59] is one of the main preprocessing and inprocessing techniques. Following Gabbay and Ohlbach's ideas [67], Khasidashvili and Korovin [91] generalized variable elimination to first-order logic with equality and demonstrated that it is effective as a preprocessor. We propose an improvement that makes this applicable in more cases and show that, with a minor restriction, it can be integrated in a superposition prover without compromising its refutational completeness.

8.4.1 Singular Predicates

Khasidashvili and Korovin's preprocessing technique removes singular predicates (which they call "non-self-referential predicates") from the problem using so-called flat resolution.

Definition 8.10. A predicate symbol is called *singular* for a clause set N if it occurs at most once in every clause contained in N.

Definition 8.11. Let $C = \mathsf{p}(\overline{s}_n) \vee C'$ and $D = \neg\mathsf{p}(\overline{t}_n) \vee D'$ be clauses with no variables in common. The clause $s_1 \not\approx t_1 \vee \cdots \vee s_n \not\approx t_n \vee C' \vee D'$ is a *flat resolvent* of C and D on p.

Given two (possibly identical) clause sets M, N, predicate elimination iteratively replaces clauses from N containing the symbol p with all flat resolvents against clauses in M. Eventually, it yields a set with no occurrences of p.

Definition 8.12. Let M, N be clause sets and p be a singular predicate for M. Let $\rightsquigarrow$ be the following relation on clause set pairs and clause sets:

1. $(M, \{(\neg)\mathsf{p}(\overline{s}) \vee C'\} \uplus N) \rightsquigarrow (M, N' \cup N)$ if N' is the set that consists of all clauses (up to variable renaming) that are flat resolvents with $(\neg)\mathsf{p}(\overline{s}) \vee C'$ on p and a clause from M as premises. The premises' variables are renamed apart.

2. $(M, N) \rightsquigarrow N$ if N has no occurrences of p.

The *resolved set* $M \bowtie_{\mathsf{p}} N$ is the clause set N' such that $(M, N) \rightsquigarrow^* N'$.

Lemma 8.13. *Let M, N be clause sets and p be a singular predicate for M. The resolved set N' is reached in a finite number of $\rightsquigarrow$ steps, and it is unique up to variable renaming.*

Proof. To show $\rightsquigarrow$ is terminating we use the following ordinal measure on clause sets: $\nu(\{D_1, \ldots, D_n\}) = \omega^{\nu(D_1)} \oplus \cdots \oplus \omega^{\nu(D_n)}$, where $\nu(D)$ is the number of occurrences of p in D, ω is the first infinite ordinal, and $\oplus$ is the Hessenberg, or natural, sum, which is commutative. For every transition $(M, \{C\} \cup N) \rightsquigarrow (M, N' \cup N)$, we have $\nu(\{C\} \cup N) > \nu(N' \cup N)$ because $\omega^{\nu(C)} > \omega^{\nu(C)-1} \cdot |N'|$. Eventually, a state (M, N') with $\nu(N') = \omega^0 \cdot n$ is reached. Then, we apply the second rule of Definition 8.12 to obtain the resolved set N'.

To show that N' is unique, i.e., $\rightsquigarrow$ is confluent, it suffices to show (since $\rightsquigarrow$ is terminating and Newmann's lemma applies [5]) that $\rightsquigarrow$ is locally confluent. In other words, whenever $(M,N)\rightsquigarrow(M,N_1)$ and $(M,N)\rightsquigarrow(M,N_2)$, there exists N' such that $(M,N_1)\rightsquigarrow(M,N')$ and $(M,N_2)\rightsquigarrow(M,N')$.

There are two main sources of nondeterminism of $\rightsquigarrow$: The choice of $C\in N$ and the choice of a literal in C to act on. Let us focus on the choice of C in N; the same discussion applies for the choice of a literal in C.

Let $N=\{C_1\}\uplus\{C_2\}\uplus N'$, where C_1 and C_2 are clauses with occurrences of p. Then, $(M,\{C_1\}\uplus\{C_2\vee N'\})\rightsquigarrow(M,N_1'\cup\{C_2\vee N'\})$ and $(M,\{C_2\}\uplus\{C_1\vee N'\})\rightsquigarrow(M,N_2'\cup\{C_1\vee N'\})$ where N_1' and N_2' are sets of corresponding resolvents. Both $\rightsquigarrow$ steps can be joined (up to variable renaming) to $(M,N_1'\cup N_2'\cup N')$, showing that $\rightsquigarrow$ is locally confluent. □

Next, it is useful to partition clause sets into subsets based on the presence and polarity of a singular predicate.

Definition 8.14. Let N be a clause set and p be a singular predicate for N. Let N_p^+ consist of all clauses of the form $\mathsf{p}(\overline{s})\vee C'\in N$, let N_p^- consist of all clauses of the form $\neg\mathsf{p}(\overline{s})\vee C'\in N$, let $N_\mathsf{p}=N_\mathsf{p}^+\cup N_\mathsf{p}^-$, and let $\overline{N}_\mathsf{p}=N\setminus N_\mathsf{p}$.

Definition 8.15. Let N be a clause set and p be a singular predicate for N. *Singular predicate elimination* (SPE) of p in N replaces N by $\overline{N}_\mathsf{p}\cup(N_\mathsf{p}^+\bowtie_\mathsf{p}N_\mathsf{p}^-)$.

The result of SPE is satisfiable if and only if N is satisfiable [91, Theorem 1], justifying SPE's use in a preprocessor. However, eliminating singular predicates aggressively can dramatically increase the number of clauses. To prevent this, Khasidashvili and Korovin suggested to replace N by N' only if $\lambda(N')\le\lambda(N)$ and $\mu(N')\le\mu(N)$, where $\lambda(N)$ is the number of literals in N and $\mu(N)$ is the sum for all clauses $C\in N$ of the square of the number of distinct variables in C.

Compared with what modern SAT solvers use, this criterion is fairly restrictive. We relax it to make it possible to eliminate more predicates, within reason. Let $K_\mathrm{tol}\in\mathbb{N}$ be a tolerance parameter. A predicate elimination step from N to N' is allowed if $\lambda(N')<\lambda(N)+K_\mathrm{tol}$ or $\mu(N')<\mu(N)$ or $|N'|<|N|+K_\mathrm{tol}$. Intuitively, we allow predicate elimination even in the cases in which proof state increases in size, as long as we can control this increase with the parameter K_tol. A refinement, which we want to try out in future work, would be to gradually increment the tolerance K_tol, as is done in some SAT solvers.

8.4.2 Defined Predicates

SPE is effective, but an important refinement has not yet been adapted to first-order logic: variable elimination by substitution. Eén and Biere [59] discovered that a propositional variable x can be eliminated without computing all resolvents if it is expressible as an equivalence $\mathsf{x}\leftrightarrow\varphi$, where φ, the "gate," is an arbitrary formula that does not reference x. They partition a set N of propositional clauses into a definition set G, essentially the clausification of $\mathsf{x}\leftrightarrow\varphi$, and $R=N_\mathsf{p}\setminus G$, the remaining clauses containing p. To eliminate x from N while preserving satisfiability, it suffices to resolve clauses from G against clauses from R, effectively substituting φ for x in R. Crucially, we do not need to resolve pairs of clauses from G or pairs of clauses from R. We generalize this idea to first-order logic.

Definition 8.16. Let G be a clause set and p be a predicate symbol. The set G is a *definition set* for p if

1. p is singular for G
2. G consists of clauses of the form $(\neg)\mathsf{p}(\overline{x}) \vee C'$ (up to variable renaming), where the variables $\overline{x}$ are distinct
3. the variables in C' are all among p's arguments $\overline{x}$
4. all clauses in $G^+_\mathsf{p} \asymp_\mathsf{p} G^-_\mathsf{p}$ are tautologies
5. $E(\overline{\mathsf{c}})$ is unsatisfiable, where the *environment* $E(\overline{x})$ consists of all subclauses C' of any $(\neg)\mathsf{p}(\overline{x}) \vee C' \in G$ and $\overline{\mathsf{c}}$ is a tuple of distinct fresh constants substituted in for $\overline{x}$

A definition set G corresponds intuitively to a definition by cases in mathematics—e.g.,

$$\mathsf{p}(\overline{x}) = \begin{cases} \top & \text{if } \varphi(\overline{x}) \\ \bot & \text{if } \psi(\overline{x}) \end{cases}$$

Part 4 states that the case conditions are mutually exclusive (e.g., $\neg\varphi(\overline{x}) \vee \neg\psi(\overline{x})$), and part 5 states that they are exhaustive (e.g., $\nexists\overline{\mathsf{c}}.\ \neg\varphi(\overline{\mathsf{c}}) \wedge \neg\psi(\overline{\mathsf{c}})$). Given a quantifier-free formula $\mathsf{p}(\overline{x}) \leftrightarrow \varphi(\overline{x})$ with distinct variables $\overline{x}$ such that $\varphi(\overline{x})$ does not contain p, any reasonable clausification algorithm would produce a definition set for p.

Example 8.17. Given the formula $\mathsf{p}(x) \leftrightarrow \mathsf{q}(x) \wedge (\mathsf{r}(x) \vee \mathsf{s}(x))$, a standard clausification algorithm [126] produces $\{\neg\mathsf{p}(x) \vee \mathsf{q}(x), \neg\mathsf{p}(x) \vee \mathsf{r}(x) \vee \mathsf{s}(x), \mathsf{p}(x) \vee \neg\mathsf{q}(x) \vee \neg\mathsf{r}(x), \mathsf{p}(x) \vee \neg\mathsf{q}(x) \vee \neg\mathsf{s}(x)\}$, which qualifies as a definition set for p.

Definition sets generalize Eén and Biere's gates. They can be recognized syntactically for formulas such as $\mathsf{p}(\overline{x}) \leftrightarrow \bigvee_i \mathsf{q}_i(\overline{s_i})$ or $\mathsf{p}(\overline{x}) \leftrightarrow \bigwedge_i \mathsf{q}_i(\overline{s_i})$, or semantically: Condition 4 can be checked using the congruence closure algorithm, and condition 5 amounts to a propositional unsatisfiability check.

The key result about propositional gates carries over to definition sets.

Definition 8.18. Let N be a clause set, p be a predicate symbol, $G \subseteq N$ be a definition set for p, and $R = N_\mathsf{p} \setminus G$. *Defined predicate elimination* (DPE) of p in N replaces N by $\overline{N}_\mathsf{p} \cup (G \asymp_\mathsf{p} R)$.

Lemma 8.19. *Let $N(\overline{x})$ be a clause set such that the variables of all clauses in it are among the argument n-tuple $\overline{x}$, and let $\overline{\mathsf{c}}$ be an n-tuple of distinct fresh constants. If $N(\overline{\mathsf{c}})$ (i.e., $\{\overline{x} \mapsto \overline{\mathsf{c}}\}(N(\overline{x}))$) is unsatisfiable, then for every interpretation $\mathcal{J}$ and valuation ξ, $\mathcal{J} \not\models_\xi N$.*

Proof. We show the contrapositive. Assume that for some $\mathcal{J}$ and ξ, $\mathcal{J} \models_\xi N(\overline{x})$. Then let $\mathcal{J}'$ be a model that assigns each c_i the interpretation of x_i under $\mathcal{J}$ and ξ, and otherwise coincides with $\mathcal{J}$. We obtain $\mathcal{J}' \models N(\overline{\mathsf{c}})$. □

Lemma 8.20. *Let G be a definition set for p and N be an arbitrary clause set. If $(G, N) \rightsquigarrow (G, N')$, then $G \cup N$ and $G \cup N'$ are equivalent.*

Proof. Since flat resolution is sound, the nontrivial direction is to show that a model $\mathcal{J}$ of the set $G \cup N'$ is also a model of $G \cup N$. As the only clause in $N \setminus N'$ is $C = (\neg)\mathsf{p}(\overline{s}_n) \vee C'$ on which the $\rightsquigarrow$ step is performed, we must show $\mathcal{J} \models C$.

Without loss of generality, we assume that the leading literal of C is positive (i.e., C is of the form $\mathsf{p}(\overline{s}_n) \vee C'$). Towards a contradiction, assume ξ is a valuation such that $\mathcal{J} \not\models_\xi C$. Then, $\mathcal{J} \not\models_\xi \mathsf{p}(\overline{s}_n)$. Consider an arbitrary clause $D = \mathsf{p}(\overline{x}_n) \vee D' \in G^+_\mathsf{p}$ and a valuation ξ', which assigns each x_i the interpretation of s_i under $\mathcal{J}$ and ξ. As $\mathcal{J} \not\models_{\xi'} \mathsf{p}(\overline{x}_n)$ and $\mathcal{J} \models G$, then $\mathcal{J} \models_{\xi'} D'$ for every such clause D. However, by part 5 of Definition 8.16 and by Lemma 8.19, $\mathcal{J} \not\models_{\xi'} E(\overline{x}_n)$, where $E(\overline{x}_n)$ is the environment associated with the definition set G. Therefore, there must exist a clause $D = \neg\mathsf{p}(\overline{x}_n) \vee D'$ in G^-_p such that $\mathcal{J} \not\models_\xi D'$.

Now consider the flat resolvent of C and D on p: $R = x_1 \not\approx s_1 \vee \cdots \vee x_n \not\approx s_n \vee C' \vee D'$. Let ζ be a valuation coinciding with ξ on the variables of C and with ξ' on $\overline{x}_n$. Clearly, $\mathcal{J} \not\models_\zeta R$. Yet, $R \in N'$, and as $\mathcal{J} \models N'$, we reach a contradiction. □

Lemma 8.21. *Let G be a definition set for p and N be a clause set with no occurrences of p. Then $G \cup N$ is satisfiable if and only if N is satisfiable.*

Proof. The nontrivial direction is to show that if N is satisfiable, $G \cup N$ is as well. Let $\mathcal{J}$ be a model of N. We construct a model $\mathcal{J}'$ of G over the same universe as $\mathcal{J}$. For any atom A such that p does not occur in A and for every ξ, we set $\mathcal{J}' \models_\xi A$ if and only if $\mathcal{J} \models_\xi A$. For any clause $\mathsf{p}(\overline{x}_n) \vee C' \in G$ and any assignment ξ such that $\mathcal{J} \not\models_\xi C'$, we define $\mathcal{J}'$ so that $\mathcal{J}' \models_\xi \mathsf{p}(\overline{x}_n)$. By construction, $\mathcal{J}' \models G^+_\mathsf{p} \cup N$. It remains to show that $\mathcal{J}' \models G^-$.

Let $C = \neg\mathsf{p}(\overline{x}_n) \vee C' \in G$ and let ξ be an arbitrary assignment. Towards a contradiction, assume $\mathcal{J}' \not\models_\xi C$, and consequently $\mathcal{J}' \models_\xi \mathsf{p}(\overline{x}_n)$. By construction of $\mathcal{J}'$, there exists a clause $\mathsf{p}(\overline{y}_n) \vee D' \in G$ and an assignment ξ' which assigns each y_i the value of $\xi(x_i)$ such that $\mathcal{J} \not\models_{\xi'} D'$. The resolvent $R = x_1 \not\approx y_1 \vee \cdots \vee x_n \not\approx y_n \vee C' \vee D'$ is a tautology, according to condition 4 of Definition 8.16. However, for a valuation that behaves like ξ on $\overline{x}$ and ξ' on $\overline{y}$, $\mathcal{J}'$ does not satisfy $R \in N$, contradicting our assumption. □

Theorem 8.22. *The result of applying DPE to a clause set N is satisfiable if and only if N is satisfiable.*

Proof. Let p be a predicate symbol and $G \subseteq N$ be the definition set used by DPE, and let $R = N_\mathsf{p} \setminus G$.

Using Lemma 8.13, we get that there is a derivation $(G, R) \rightsquigarrow^n (G, R') \rightsquigarrow R'$. Applying Lemma 8.20 n times, we get that $G \cup R$ is equivalent to $G \cup R'$. Finally, Lemma 8.21 gives us the desired result. □

Since there will typically be at most only a few defined predicates in the problem, it makes sense to fall back on SPE when no definition is found.

Definition 8.23. Let N be a clause set and p be a predicate symbol. If there exists a definition set $G \subseteq N$ for p, *portfolio predicate elimination* (PPE) on p in N replaces N with $\overline{N}_\mathsf{p} \cup (G \bowtie_\mathsf{p} R)$, where $R = N_\mathsf{p} \setminus G$. Otherwise, if p is singular in N, it results in $\overline{N}_\mathsf{p} \cup (N^+_\mathsf{p} \bowtie_\mathsf{p} N^-_\mathsf{p})$. In all other cases, it is not applicable.

8.4.3 Refutational Completeness

Hidden-literal-based techniques fit within the traditional framework of saturation, because they delete or reduce a clause based on the *presence* of other clauses. In contrast, predicate elimination relies on the *absence* of clauses from the proof state. We can still integrate it with superposition as follows: At every kth iteration of the given clause procedure, perform predicate elimination on $\mathscr{A} \cup \mathscr{P}$, and add all new clauses to $\mathscr{P}$.

One may wonder whether such an approach preserves the refutational completeness of the calculus. The answer is no.

To see why, consider the following *binary splitting* rule based on Riazanov and Voronkov [139]:

$$\frac{C \vee D}{\mathsf{p} \vee C \quad D \vee \neg \mathsf{p}}\,\mathrm{BS}$$

Provisos: C and D have no free variables in common, p is fresh, and p is $\prec$-smaller than C and D. Since the conclusions are smaller than the premise, the rule can be applied aggressively as a simplification. But notice that the effect of splitting can be undone by singular predicate elimination, possibly giving rise to loops $\mathrm{BS}, \mathrm{SPE}, \mathrm{BS}, \mathrm{SPE}, \ldots$. Clearly, we need to curtail predicate elimination.

Under which conditions is predicate elimination refutationally complete? To answer this question, we employ the saturation framework of Waldmann, Tourret, Robillard, and Blanchette [171]. Let $(\mathit{FInf}, \mathit{Red})$ be the base calculus without predicate elimination—e.g., resolution or superposition inferences together with the standard redundancy criterion [8, Sect. 4.2]. The inference system FInf is a set of inferences $(C_n, \ldots, C_1, C_0)$, for $n \geq 1$, where $C_n, \ldots, C_1$ are the premises and C_0 is the conclusion. C_1 is called the main premise. The redundancy criterion is a pair $\mathit{Red} = (\mathit{Red}_\mathrm{I}, \mathit{Red}_\mathrm{F})$ where Red_I determines which inferences can be omitted and Red_F is used to remove clauses.

Next, consider an abstract proving process working on a single clause set. Let $\rhd_{\mathit{Red}}$ denote the transition relation that supports (1) adding arbitrary clauses and (2) removing clauses deemed useless by Red_F. Typically, the added clauses are the result of performing inferences and are entailed by the premises, but other clauses can be added as well. A $\rhd_{\mathit{Red}}$*-derivation* is a finite or infinite sequence of clause sets $N_0 \rhd_{\mathit{Red}} N_1 \rhd_{\mathit{Red}} \cdots$.

We fix a finite set $\mathbf{P}$ of predicate symbols that may be subjected to predicate elimination. These might include all the predicate symbols occurring in the input problem, but exclude any symbols introduced by splitting or other rules. Given a clause or clause set N, we write $\mathbf{P}(N)$ to denote the set of all predicate symbols from $\mathbf{P}$ occurring in N. Let $\rhd_{\mathbf{P}}$ denote the elimination of a singular or defined predicate symbol from $\mathbf{P}$. A *mixed derivation* consists of transitions either of the form $N \rhd_{\mathbf{P}} N'$ or of the form $N \rhd_{\mathit{Red}} N'$ where $\mathbf{P}(N) \supseteq \mathbf{P}(N')$. Because $\mathbf{P}$ is finite, any mixed derivation consists of at most finitely many $\rhd_{\mathbf{P}}$-transitions. Hence, in any derivation, there exists an index k from which all transitions are standard $\rhd_{\mathit{Red}}$-transitions.

This suggests the following path to completeness: Pretend that the transitions between N_0 and N_k are merely preprocessing and start the actual derivation at N_k. This works at the abstract level of derivations on single clause sets. It fails, however, for an actual saturation prover that distinguishes between passive and active clauses.

Example 8.24. The counterexample below is based on the given clause prover GC from the saturation framework. It shows how predicate elimination can break GC's key invariant, which states that all inferences between active clauses are redundant. Breaking the invariant means that the limit might be unsaturated, breaking the refutational completeness proof.

We use superposition with the order $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{d}$ and without selection. Assume $\mathsf{a} \in \mathbf{P}$ and suppose we start with the satisfiable clause set

$$\neg \mathsf{a} \vee \boxed{\mathsf{d}} \qquad \neg \mathsf{a} \vee \boxed{\neg \mathsf{d}} \qquad \mathsf{a} \vee \mathsf{b} \vee \boxed{\mathsf{c}} \qquad \mathsf{c} \vee \boxed{\mathsf{d}} \qquad \mathsf{b} \vee \boxed{\neg \mathsf{d}}$$

where gray boxes mark maximal (i.e., eligible) literals. Suppose the prover makes $\mathsf{c} \vee \mathsf{d}$ and $\mathsf{b} \vee \neg \mathsf{d}$ active. From these two clauses, a superposition inference ι could derive the conclusion $\mathsf{b} \vee \mathsf{c}$. However, the three passive clauses are $\prec$-smaller than ι's main premise $\mathsf{b} \vee \neg \mathsf{d}$ and collectively entail ι's conclusion. This means that ι is redundant and can be ignored.

If the prover now eliminates the predicate a using SPE, the passive set is reduced to $\{\mathsf{b} \vee \mathsf{c} \vee \mathsf{d},\, \mathsf{b} \vee \mathsf{c} \vee \neg \mathsf{d}\}$. Either clause is subsumed by an active clause, so the prover deletes it. It stops with the active set $\{\mathsf{c} \vee \mathsf{d},\, \mathsf{b} \vee \neg \mathsf{d}\}$, which is unsaturated because ι is no longer redundant. The invariant is broken.

Example 8.25. In Example 8.24, the initial clause set was satisfiable. If it is unsatisfiable, we can even lose refutational completeness. To see why, we add the unit clauses $\neg \mathsf{b}$ and $\neg \mathsf{c}$ to the initial clause set of Example 8.24 to make it unsatisfiable. We repeat the same steps as above, including the subsumptions at the end, yielding the passive set $\{\boxed{\neg \mathsf{b}}, \boxed{\neg \mathsf{c}}\}$ and the active set $\{\mathsf{c} \vee \boxed{\mathsf{d}}, \mathsf{b} \vee \boxed{\neg \mathsf{d}}\}$. Then, making $\neg \mathsf{b}$ and $\neg \mathsf{c}$ active triggers no inferences. The prover stops with an unsatisfiable four-clause active set that does not contain the empty clause.

A solution could be to move all active clauses to the passive set at step k or later, but this would be costly, since it would force the prover to redo inferences whose conclusions might then have to be simplified or subsumed again. Instead, we salvage the existing completeness proof for GC, by resolving the issues concerning splitting and the GC invariant. Our approach is to weaken the redundancy criterion slightly, enough both to disable splitting on **P**-predicates and to ensure that inferences such as ι in Examples 8.24 and 8.25 are performed. The required weakening is so mild that it does not invalidate any practical simplification or subsumption techniques we are aware of, except of course splitting.

In accordance with the saturation framework, let **F** be the set of first-order Σ-clauses, let **G** be its ground subset, and let $\mathcal{G}$ be a function that maps an **F**-clause to the set of its **G**-clause instances and analogously for **F**-inferences. We define an extension $\mathbf{G}^\flat$ of **G** for $\Sigma^\flat$-clauses in an ad hoc nonclassical logic reminiscent of paraconsistent logics [43]. The objective is to disallow the entailment that makes splitting and Examples 8.24 and 8.25 possible. The signature $\Sigma^\flat$ extends Σ with a distinguished predicate symbol $\bot$ that is interpreted differently from $\top$. For $\Sigma^\flat$, the Boolean type o may be interpreted as any set of cardinality at least 2.

Definition 8.26. The operator $^\flat$ translates Σ-literals to $\Sigma^\flat$-literals as follows, where $\mathsf{p} \in \mathbf{P}$, $\mathsf{q} \notin \mathbf{P}$, and s, t are non-Boolean terms:

$$\mathsf{p}(\bar{t})^\flat = \mathsf{p}(\bar{t}) \not\approx \bot \qquad \mathsf{q}(\bar{t})^\flat = \mathsf{q}(\bar{t}) \approx \top \qquad (s \approx t)^\flat = s \approx t$$
$$\neg\mathsf{p}(\bar{t})^\flat = \mathsf{p}(\bar{t}) \not\approx \top \qquad \neg\mathsf{q}(\bar{t})^\flat = \mathsf{q}(\bar{t}) \not\approx \top \qquad (s \not\approx t)^\flat = s \not\approx t$$

The operator is lifted elementwise to **G**-clauses and **G**-clause sets. The *weak entailment* $\models^\flat$ over **G**-clause sets is defined via an encoding into $\Sigma^\flat$-clauses: $M \models^\flat N$ if and only if $M^\flat \models N^\flat$. The lifting to **F**-clauses and **F**-clause set is achieved in the standard way via grounding.

The following property of weak entailment will allow us to eliminate **P**-predicates without losing completeness:

Lemma 8.27. *Let C be a clause that contains the predicate symbol $\mathsf{p} \in \mathbf{P}$ and D be a clause that does not contain p. If $N \cup \{C\} \models^\flat \{D\}$, then $N \models^\flat \{D\}$.*

Proof. Suppose $\mathcal{J} \models N^\flat$. We will define $\mathcal{J}'$ so that $\mathcal{J}' \models N^\flat \cup \{C^\flat\}$, retrieve $\mathcal{J}' \models D^\flat$, and then argue that $\mathcal{J} \models D^\flat$. We take $\mathcal{J}'$ to coincide with $\mathcal{J}$ except that we extend the domain for o with one fresh value and use this value as the interpretation of $\mathsf{p}(\bar{t})$ for all argument tuples $\bar{t}$. This modification makes any p literal of $C^\flat$ true, and it preserves the truth of $N^\flat$. By the hypothesis, $\mathcal{J}' \models D^\flat$. And since p does not occur in D, we have $\mathcal{J} \models D^\flat$. □

Note that the above lemma does not hold for classical entailment $\models$; indeed, $\{\mathsf{p} \vee \mathsf{q}, \neg\mathsf{p} \vee \mathsf{q}\} \models \{\mathsf{q}\}$ yet $\{\mathsf{p} \vee \mathsf{q}\} \not\models \{\mathsf{q}\}$. On the other hand, the law of excluded middle does hold for weak entailment: $\models^\flat \mathsf{p} \vee \neg\mathsf{p}$. In fact, all classical clausal tautologies are tautologies for $\models^\flat$.

The standard redundancy criterion *Red* is obtained by lifting a criterion on **G**-clauses to **F**-clauses. The same construction can be replicated using $\models^\flat$ instead of $\models$, yielding the *weak redundancy criterion* $Red^\flat$. It is easy to check that the usual simplification techniques implemented in superposition provers can be justified using $Red^\flat$. Specifically, this concerns the following rules described by Schulz [143, Sects. 2.3.1 and 2.3.2]: deletion of duplicated literals, deletion of resolved literals, syntactic tautology deletion, semantic tautology deletion, rewriting of negative literals, positive simplify-reflect, negative simplify-reflect, clause subsumption, and equality subsumption. Moreover, rewriting of positive literals is possible if the rewriting clause is smaller than the rewritten clause (a condition that is also needed with $\models$ but omitted by Schulz to allow more aggressive application of this useful rule). Finally, destructive equality resolution cannot be justified with $\models$, let alone $\models^\flat$.

We instantiate the saturation framework with $(FInf, Red^\flat)$ to obtain a given clause prover GC. The prover operates on sets of labeled clauses (C, l), where C is a standard clause and $l \in \mathbf{L}$ is a label. The active label identifies active clauses; all other clauses are passive. The prover takes the form of two rules, Process and Infer, restricted to prevent the introduction of **P**-predicates. We extend it with a third rule, PredElim, for predicate elimination, and call the extended prover GCP. The rules are as follows, using again the framework notations:

Process $\quad \mathcal{N} \cup \mathcal{M} \Longrightarrow_{\mathsf{GCP}} \mathcal{N} \cup \mathcal{M}'$
where $\mathcal{M} \subseteq LRed_{\mathrm{F}}^{\flat,\sqsupset}(\mathcal{N} \cup \mathcal{M}')$, $\mathcal{M}'\downarrow_{\mathsf{active}} = \emptyset$, and
$\mathbf{P}(\lfloor \mathcal{M}' \rfloor) \subseteq \mathbf{P}(\lfloor \mathcal{N} \cup \mathcal{M} \rfloor)$

Infer $\quad \mathcal{N} \cup \{(C, l)\} \Longrightarrow_{\mathsf{GCP}} \mathcal{N} \cup \{(C, \mathsf{active})\} \cup \mathcal{M}$
where $l \neq \mathsf{active}$, $\mathcal{M}\downarrow_{\mathsf{active}} = \emptyset$,

$FInf(\lfloor \mathcal{N}\downarrow_{\mathsf{active}} \rfloor, \{C\}) \subseteq Red_{\mathrm{I}}^{\flat} \cap^{\mathcal{G}} (\lfloor \mathcal{N} \rfloor \cup \{C\} \cup \lfloor \mathcal{M} \rfloor)$, and
$\mathbf{P}(\lfloor \mathcal{M} \rfloor) \subseteq \mathbf{P}(\lfloor \mathcal{N} \rfloor \cup \{C\})$

PredElim $\mathcal{N} \cup \mathcal{M} \Longrightarrow_{\mathsf{GCP}} \mathcal{N} \cup \mathcal{M}'$
where $\mathcal{N} \cup \mathcal{M} \rhd_{\mathbf{P}} \mathcal{N} \cup \mathcal{M}'$ and $\mathcal{M}'\downarrow_{\mathsf{active}} = \emptyset$

Here is a summary of the main framework notations:

- The letters $\mathcal{M}, \mathcal{N}$ range over sets of labeled clauses. $\mathcal{M}\downarrow_l$ denotes the subset of clauses in $\mathcal{M}$ labeled with l. The operator $\lfloor\ \rfloor$ erases all labels in a labeled clause or clause set.

- $FInf(N)$ denotes the set of all base calculus inferences with premises in N, and $FInf(N, M) = FInf(N \cup M) \setminus FInf(N \setminus M)$. The same notations are also available for the straightforward extension *FLInf* of *FInf* with labels.

- $LRed^{\flat,\sqsupset}$ is the extension of the standard redundancy criterion $Red^{\flat}$ defined using $\models^{\flat}$ to nonground labeled clauses with subsumption ($\sqsubset$).

- Given a sequence $(\mathcal{N}_i)_i$, its *limit* (*inferior*) is $\mathcal{N}_\infty = \bigcup_i \bigcap_{j \geq i} \mathcal{N}_j$.

The completeness proof follows the invariance-based argument found in the article by Waldmann et al. [172] about the saturation framework.

Lemma 8.28. *Every $\Longrightarrow_{\mathsf{GCP}}$-derivation is a mixed derivation.*

Proof. The cases for Process and Infer are almost as in Waldmann et al., with adjustments to show that P-predicates cannot appear from nowhere. The case for ElimPred is trivial. □

Let $Inv_{\mathcal{N}}^{\flat}(k)$ denote the condition $FLInf(\mathcal{N}_k\downarrow_{\mathsf{active}}) \subseteq LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_k)$. Notice the difference with the definition of the key invariant $Inv_{\mathcal{N}}$ in the saturation framework, whose right-hand side is $\bigcup_{i=0}^{k} Red_{\mathrm{I}}^{\mathbf{L}}(\mathcal{N}_i)$. We cannot use the big union $\bigcup$ starting at $i = 0$ because we will need to truncate a sequence prefix of an a priori unknown length. The argument will still work thanks to monotonicity properties of the redundancy criteria.

Lemma 8.29. *Let $(\mathcal{N}_i)_i$ be a $\Longrightarrow_{\mathsf{GCP}}$-derivation. If $\mathcal{N}_0\downarrow_{\mathsf{active}} = \emptyset$, then $Inv_{\mathcal{N}}^{\flat}(k)$ holds for all indices k.*

Proof. The base case is as in Waldmann et al.

For Process and Infer, the proof is essentially as in Waldmann et al., except that we also need to show that $LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_k) \subseteq LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_{k+1})$. This is a consequence of $\mathcal{N}_k \rhd_{LRed^{\flat,\sqsupset}} \mathcal{N}_{k+1}$ and of properties (R2) and (R3) of redundancy criteria.

A new case to consider is that of a PredElim transition $\mathcal{N}_k \Longrightarrow_{\mathsf{GCP}} \mathcal{N}_{k+1}$. Let $\mathcal{N}_k = \mathcal{N} \cup \mathcal{M} \Longrightarrow_{\mathsf{GCP}} \mathcal{N} \cup \mathcal{M}' = \mathcal{N}_{k+1}$, where $\mathcal{N} \cup \mathcal{M} \rhd_{\mathbf{P}} \mathcal{N} \cap \mathcal{M}'$ and $\mathcal{M}'\downarrow_{\mathsf{active}} = \emptyset$. We assume without loss of generality that $\mathcal{M} \cap \mathcal{M}' = \emptyset$. Let p be the eliminated predicate. Note that p occurs in every clause in $\mathcal{M}$ but in none of the clauses in $\mathcal{N}$ or $\mathcal{M}'$. We must show $FLInf(\mathcal{N}_{k+1}\downarrow_{\mathsf{active}}) \subseteq LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_{k+1})$. As for Process, we have the inclusion

$FLInf(\mathcal{N}_{k+1}{\downarrow_{\mathsf{active}}}) \subseteq FLInf(\mathcal{N}_k{\downarrow_{\mathsf{active}}})$, by the side condition that $\mathcal{M}'{\downarrow_{\mathsf{active}}} = \emptyset$. Moreover, by the induction hypothesis, $FLInf(\mathcal{N}_k{\downarrow_{\mathsf{active}}}) \subseteq LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_k)$. Thus, $FLInf(\mathcal{N}_k{\downarrow_{\mathsf{active}}}) \subseteq LRed_{\mathrm{I}}^{\flat}(\mathcal{N} \cup \mathcal{M})$.

Let $\iota \in FLInf(\mathcal{N}{\downarrow_{\mathsf{active}}})$. By the argument above, we have $\iota \in LRed_{\mathrm{I}}^{\flat}(\mathcal{N} \cup \mathcal{M})$. We must show $\iota \in LRed_{\mathrm{I}}^{\flat}(\mathcal{N} \cup \mathcal{M}')$. By definition of $LRed_{\mathrm{I}}^{\flat}$, it suffices to show that for every ground instance $(C_n, \ldots, C_1, C_0)$ of ι, there exists a finite clause set $\mathcal{D} \subseteq \mathcal{G}(\mathcal{N}) \cup \mathcal{G}(\mathcal{M}')$ that is $\prec$-smaller than C_1 and such that $\{C_n, \ldots, C_2\} \cup \mathcal{D} \models^{\flat} \{C_0\}$. Without loss of generality, we assume that $\mathcal{D}$ is the $\prec$-smallest such set; such a set exists because $\prec$ is well founded.

By definition of $\mathcal{N}$, p cannot occur in C_0. By Lemma 8.27, if p occurred in $D \in \mathcal{D}$, we could remove D, but this would mean $\mathcal{D}$ is not minimal. As a result, $\mathcal{D}$ cannot contain clauses from $\mathcal{G}(\mathcal{M})$ and hence $\mathcal{D} \subseteq \mathcal{G}(\mathcal{N})$. Thus, $\iota \in LRed_{\mathrm{I}}^{\flat}(\mathcal{N})$. By property (R2) of redundancy criteria, we have the desired result: $\iota \in LRed_{\mathrm{I}}^{\flat}(\mathcal{N} \cup \mathcal{M}')$. □

Lemma 8.30. *Let $(\mathcal{N}_i)_i$ be a $\rhd_{LRed^{\flat,\sqsupset}}$-derivation. If $Inv_{\mathcal{N}}^{\flat}(i)$ holds for all indices i, then $FLInf(\mathcal{N}_\infty{\downarrow_{\mathsf{active}}}) \subseteq \bigcup_i LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_i)$ holds.*

Proof. We assume $\iota \in FLInf(\mathcal{N}_\infty{\downarrow_{\mathsf{active}}})$ and show $\iota \in \bigcup_i LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_i)$ for some arbitrary ι. For ι to belong to $FLInf(\mathcal{N}_\infty{\downarrow_{\mathsf{active}}})$, all of its finitely many premises must be in $\mathcal{N}_\infty{\downarrow_{\mathsf{active}}}$. Therefore, there must exist an index k such that $\mathcal{N}_k{\downarrow_{\mathsf{active}}}$ contains all of them, and therefore $\iota \in FLInf(\mathcal{N}_k{\downarrow_{\mathsf{active}}})$. Since $Inv_{\mathcal{N}}(k)$ holds, $\iota \in LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_k)$. Hence, $\iota \in \bigcup_i LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_i)$. □

Lemma 8.31. *Let $(\mathcal{N}_i)_i$ be a $\Longrightarrow_{\mathsf{GCP}}$-derivation. If $\mathcal{N}_0{\downarrow_{\mathsf{active}}} = \emptyset$ and $\mathcal{N}_\infty{\downarrow_l} = \emptyset$ for every label $l \neq \mathsf{active}$, then there exists an index k such that $(\mathcal{N}_{k+i})_i$ is a fair $\rhd_{LRed^{\flat,\sqsupset}}$-derivation.*

Proof. By Lemma 8.28, there must exist an index k such that the sequence $(\mathcal{N}_{k+i})_i$ is a pure $\rhd_{LRed^{\flat,\sqsupset}}$-derivation. By Lemma 8.29, $Inv_{\mathcal{N}}^{\flat}(k+i)$ holds for all indices i. Hence, by Lemma 8.30, $FLInf(\mathcal{N}_\infty{\downarrow_{\mathsf{active}}}) \subseteq \bigcup_i LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_{k+i})$. By the second hypothesis, this inclusion simplifies to $FLInf(\mathcal{N}_\infty) \subseteq \bigcup_i LRed_{\mathrm{I}}^{\flat}(\mathcal{N}_{k+i})$. □

Theorem 8.32. *Let $(\mathcal{N}_i)_i$ be a $\Longrightarrow_{\mathsf{GCP}}$-derivation with $\mathcal{N}_0{\downarrow_{\mathsf{active}}} = \emptyset$ and $\mathcal{N}_\infty{\downarrow_l} = \emptyset$ for every label $l \neq \mathsf{active}$. If $\lfloor \mathcal{N}_0 \rfloor$ is unsatisfiable, then some $\mathcal{N}_i$ contains the empty clause with some arbitrary label.*

Proof. By Lemma 8.31, we know that there exists an index k such that $(\mathcal{N}_{k+i})_i$ is a fair $\rhd_{LRed^{\flat,\sqsupset}}$-derivation. Moreover, since $\rhd_{LRed^{\flat,\sqsupset}}$ and $\rhd_{\mathbf{p}}$ preserve unsatisfiability (by (R1) of redundancy criteria, Khasidashvili and Korovin's Theorem 1, and our Theorem 8.22), we have that $\lfloor \mathcal{N}_k \rfloor$ is unsatisfiable. Since the base calculus $(FLInf, LRed)$ is assumed to be statically refutationally complete with respect to $\models$, the calculus $(FLInf, LRed^{\flat})$ with a weaker redundancy criterion is also statically complete with respect to $\models$, and by the saturation framework, $(FLInf, LRed^{\flat,\sqsupset})$ preserves this. Exploiting the equivalence of static and dynamic completeness (Lemmas 10 and 11 in Waldmann et al. [172]), we conclude that some $\mathcal{N}_{k+i}$ must contain a labeled empty clause. □

8.5 Satisfiability by Clause Elimination

The main approaches to show satisfiability of a first-order problem are to produce either a finite Herbrand model or a saturated clause set. Saturations rarely occur except for very small problems or within decidable fragments. In this section, we explore an alternative approach that establishes satisfiability by iteratively removing clauses while preserving unsatisfiability, until the clause set has been transformed into the empty set. So far, this technique has been studied only for QBF [75]. We show that *blocked clause elimination* (BCE) can be used for this purpose. It can efficiently solve some problems for which the saturated set would be infinite. However, it can break the refutational completeness of a saturation prover. We conclude with a procedure that transforms a finite Herbrand model into a sequence of clause elimination steps ending in the empty clause set, thereby demonstrating the theoretical power of clause elimination.

Kiesl et al. [93] generalized blocked clause elimination to first-order logic. Their generalization uses flat L-resolvents, an extension of flat resolvents that resolves a single literal L against m literals of the other clause.

Definition 8.33. Let $C = L \vee C'$ and $D = L_1 \vee \cdots \vee L_m \vee D'$, where

1. $m \geq 1$
2. the literals L_i are of opposite polarity to L
3. L's atom is $\mathsf{p}(\overline{s}_n)$
4. L_i's atom is $\mathsf{p}(\overline{t_i})$ for each i
5. C and D have no variables in common

The clause $\left(\bigvee_{i=1}^{m} \bigvee_{j=1}^{n} s_j \not\approx t_{ij}\right) \vee C' \vee D'$ is a *flat L-resolvent* of C and D.

Definition 8.34. Let $C = L \vee C'$ be a clause and N be a clause set. Let N' consist of all clauses from $N \setminus \{C\}$ with their variables renamed so that they share no variables with C. The clause C is (*equality-*)*blocked* by L in the set N if all flat L-resolvents between C and clauses in N' are tautologies.

Removing a blocked clause from a set preserves unsatisfiability [93]. Kiesl et al. evaluated the effect of removing all blocked clauses as a preprocessing step and found that it increases the prover's success rate.

In fact, there exist satisfiable problems that cannot be saturated in finitely many steps regardless of the calculus's parameters but that can be reduced to an empty, vacuously satisfiable problem through blocked clause elimination.

Example 8.35. Consider the clause set N consisting of $C = \mathsf{p}(x, x)$ and $D = \neg\mathsf{p}(y_1, y_3) \vee \mathsf{p}(y_1, y_2) \vee \mathsf{p}(y_2, y_3)$. Note that if no literal is selected, all literals can take part in superposition inferences (i.e., they are eligible), as literals in clause D are $\succ$-uncomparable. In particular, the superposition of $\mathsf{p}(x, x)$ into D's negative literal eventually needs to be performed regardless of the chosen selection function or term order, with the conclusion $E_1 = \mathsf{p}(z_1, z_2) \vee \mathsf{p}(z_2, z_1)$. Then, superposition of E_1 into D yields $E_2 = \mathsf{p}(z_1, z_2) \vee \mathsf{p}(z_2, z_3) \vee \mathsf{p}(z_3, z_1)$. Repeating this process yields infinitely many clauses $E_i = \mathsf{p}(z_1, z_2) \vee$

$\cdots \vee \mathsf{p}(z_i, z_{i+1}) \vee \mathsf{p}(z_{i+1}, z_1)$ that cannot be eliminated using standard redundancy-based techniques.

In the example above, the clause D is blocked by its second or third literal. If we delete D, C becomes blocked in turn. Deleting C leaves us with the empty set, which is vacuously satisfiable. The example suggests that using BCE during saturation might help focus the proof search. Indeed, Kiesl et al. ended their investigations by asking whether BCE can be used as an inprocessing technique in a saturation prover. Unfortunately, in general the answer is no:

Example 8.36. Consider the unsatisfiable set $N = \{C_1, \ldots, C_6\}$, where

$$C_1 = \neg\mathsf{c} \vee \mathsf{e} \vee \neg\mathsf{a} \qquad C_2 = \neg\mathsf{c} \vee \neg\mathsf{e} \qquad C_3 = \mathsf{b} \vee \mathsf{c}$$
$$C_4 = \neg\mathsf{b} \vee \neg\mathsf{c} \qquad C_5 = \mathsf{a} \vee \mathsf{b} \qquad C_6 = \mathsf{c} \vee \neg\mathsf{b}$$

Assume the simplification ordering $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{d} \prec \mathsf{e}$ and the selection function that chooses the last negative literal of a clause as presented. Gray boxes indicate literals that can take part in superposition inferences. Only two superposition inferences are possible: from C_3 into C_4, yielding the tautology $C_7 = \mathsf{b} \vee \neg\mathsf{b}$, and from C_5 into C_6, yielding $C_8 = \mathsf{a} \vee \mathsf{c}$. Clause C_7 is clearly redundant, whereas C_8 is blocked by its first literal. If we allow removing blocked clauses, the prover enters a loop: C_8 is repeatedly generated and deleted. Thus, the prover will never generate the empty clause for this unsatisfiable set.

As with hidden tautologies, removing blocked clauses breaks the invariant of the given clause procedure that all inferences between clauses in $\mathscr{A}$ are redundant. To see this, assume the setting of Example 8.36, and let $\mathscr{P} = N$ and $\mathscr{A} = \emptyset$. Assume C_1, C_2, C_3 are moved to the active set. As there are no possible inferences between them, the proof state becomes $\mathscr{A} = \{C_1, C_2, C_3\}$ and $\mathscr{P} = \{C_4, C_5, C_6\}$. After C_4 is moved to $\mathscr{A}$, the conclusion C_7 is computed, but it is not added to $\mathscr{P}$ as it is redundant. Moving C_5 to $\mathscr{A}$ produces no new conclusions, but after C_6 is moved, C_8 is produced. However, if we allow eliminating blocked clauses, it will not be added to $\mathscr{P}$ as it is blocked. The prover then terminates with $\mathscr{A} = N$ and $\mathscr{P} = \emptyset$, even though the original set N is unsatisfiable.

Although using BCE as inprocessing breaks the completeness of superposition in general, it is conceivable that a well-behaved fragment of BCE might exist. This could be investigated further.

Not only can BCE prevent infinite saturation (Example 8.35), but it can also be used to convert a finite Herbrand model into a certificate of clause set satisfiability (i.e., an object that carries the checkable proof of satisfiability). The certificate uses only blocked clause elimination and addition, in conjunction with a transformation to reduce the clause set to an empty set. This theoretical result explores the relationship between Herbrand models and satisfiability certificates based on clause elimination and addition. It is conceivable that it can form the basis of an efficient way to certify Herbrand models.

In propositional logic, *asymmetric literals* can be added to or removed from clauses, retaining the equivalence of the resulting clause set with the original one. Kiesl and Suda [92] described an extension of this technique to first-order logic. Their definition of asymmetric literals can be relaxed to allow the addition of more literals, but the resulting set is then only equisatisfiable to the original one, not equivalent. This in turn allows

us to show that a problem is satisfiable by reducing it to an empty problem, as is done in some SAT solvers.

For the rest of this section, we work with clausal first-order logic without equality. We use Herbrand models as canonical representatives of first-order models, recalling that every satisfiable set has a Herbrand model [65, Sect. 5.4].

Definition 8.37. A literal L is a *global asymmetric literal* (GAL) for a clause C and a clause set N if for every ground instance $\sigma(C)$ of C, there exists a ground instance $\varrho(D) \vee \varrho(L')$ of $D \vee L' \in N \setminus \{C\}$ such that $\varrho(D) \subseteq \sigma(C)$ and $\varrho(\neg L') = \sigma(L)$.

Every asymmetric literal is GAL, but the converse does not hold:

Example 8.38. Consider a clause $C = \mathsf{p}(x, y)$ and a clause set $N = \{\mathsf{q} \vee \mathsf{p}(\mathsf{a}, \mathsf{a})\}$. Then, $\neg\mathsf{q}$ is not an asymmetric literal for C and N, but it is a GAL for C and N.

Adding and removing GALs preserves and reflects satisfiability:

Theorem 8.39. *If L is a GAL for the clause C and the clause set N, then the set $(N \setminus \{C\}) \cup \{C \vee L\}$ is satisfiable if and only if N is satisfiable.*

Proof. Let $N' = N \setminus \{C\} \cup \{C \vee L\}$. The nontrivial direction is to prove that if N' has a model, so does N. If N' has a model, it has an Herbrand model $\mathcal{J}$. Clearly, $\mathcal{J}$ satisfies every clause in N, with the possible exception of C. Assume that there exists a grounding substitution σ such that ground instance $\sigma(C)$ is falsified by $\mathcal{J}$. Since L is a GAL for C and N, then there exists a clause $D \vee L' \in N'$, and a grounding substitution τ such that $\tau(D) \subseteq \sigma(C)$ and $\tau(\neg L') = \sigma(L)$. If $\mathcal{J} \models \tau(L')$, for $\sigma(C) \vee \sigma(L)$ to be satisfied by $\mathcal{J}$ (as $C \vee L \in N'$), some literal in $\sigma(C)$ must be satisfied by $\mathcal{J}$, contradicting that $\sigma(C)$ is falsified by $\mathcal{J}$. If $\mathcal{J} \not\models \tau(L')$, then some literal in $\tau(D)$ must be satisfied. Since $\tau(D) \subseteq \sigma(C)$, we get the same contradiction. Therefore, $\mathcal{J}$ satisfies N, as needed. □

For first-order logic without equality, a clause $L \vee C$ is blocked if all its L-resolvents are tautologies [93]. The L-resolvent between $L \vee C$ and $\neg L_1 \vee \cdots \vee \neg L_n \vee D$ (renamed apart) is $\sigma(C \vee D)$, where σ is the MGU of the literals $L, L_1, \ldots, L_n$. Given a Herbrand model $\mathcal{J}$ of a clause set N, the following procedure removes all clauses while preserving satisfiability:

1. Let q be a fresh predicate symbol. For each atom $\mathsf{p}(\overline{s})$ in the Herbrand universe: If $\mathcal{J} \models \mathsf{p}(\overline{s})$, add the clause $\mathsf{q} \vee \mathsf{p}(\overline{s})$; otherwise, add $\mathsf{q} \vee \neg\mathsf{p}(\overline{s})$. Adding either clause preserves satisfiability as both are blocked by q.

2. Since $\mathcal{J}$ is a model, for each ground instance $\sigma(C)$, there exists a clause $\mathsf{q} \vee L$ with $L \in \sigma(C)$. We can transform $C \in N$ into $C \vee \neg\mathsf{q}$, since $\neg\mathsf{q}$ is a GAL for C and N.

3. Consider the clause $\mathsf{q} \vee L$ added by step 1. Since L is ground and no clause $\mathsf{q} \vee \neg L$ was added (since $\mathcal{J}$ is a model), the only L-resolvents are against clauses added by step 2. Since all of those clauses contain $\neg\mathsf{q}$, the resolvents are tautologies. Thus, each $\mathsf{q} \vee L$ is blocked and can be removed in turn.

4. The remaining clauses all contain the literal $\neg\mathsf{q}$. They can be removed by BCE as well.

The procedure is limited to first-order logic without equality, since step 3 is justified only if L is a predicate literal. (Otherwise, L cannot block clause $\mathsf{q} \vee L$ [93].) The procedure also terminates only for finite Herbrand models.

Example 8.40. Consider the satisfiable clause set $N = \{\mathsf{r}(x) \vee \mathsf{s}(x), \neg\mathsf{r}(\mathsf{a}), \neg\mathsf{s}(\mathsf{b})\}$ and a Herbrand model $\mathcal{J}$ over $\{\mathsf{a}, \mathsf{b}, \mathsf{r}, \mathsf{s}\}$ such that $\mathsf{r}(\mathsf{b})$ and $\mathsf{s}(\mathsf{a})$ are the only true atoms in $\mathcal{J}$. We show how to remove all clauses in N using $\mathcal{J}$ by following the procedure above.

Let $N_{\mathcal{J}} = \{\mathsf{q} \vee \neg\mathsf{r}(\mathsf{a}), \mathsf{q} \vee \mathsf{r}(\mathsf{b}), \mathsf{q} \vee \mathsf{s}(\mathsf{a}), \mathsf{q} \vee \neg\mathsf{s}(\mathsf{b})\}$. We set $N \leftarrow N \cup N_{\mathcal{J}}$. This preserves satisfiability since all clauses in $N_{\mathcal{J}}$ are blocked. It is easy to check that $\neg\mathsf{q}$ is GAL for every clause in $N \setminus N_{\mathcal{J}}$. The only substitutions that need to be considered are $\{x \mapsto \mathsf{a}\}$ and $\{x \mapsto \mathsf{b}\}$ for $\mathsf{r}(x) \vee \mathsf{s}(x)$. So we set $N \leftarrow \{\neg\mathsf{q} \vee \mathsf{r}(x) \vee \mathsf{s}(x), \neg\mathsf{q} \vee \neg\mathsf{r}(\mathsf{a}), \neg\mathsf{q} \vee \neg\mathsf{s}(\mathsf{b})\} \cup N_{\mathcal{J}}$. Clearly, all clauses in $N_{\mathcal{J}}$ are blocked, so we set $N \leftarrow N \setminus N_{\mathcal{J}}$. All clauses remaining in N have a literal $\neg\mathsf{q}$ and can be removed, leaving N empty as desired.

8.6 Implementation

Hidden-literal-based elimination, predicate elimination, and blocked clause elimination all admit efficient implementations in a superposition prover. In this section, we describe how to implement the first two sets of techniques. For BCE, we refer to Kiesl et al. [93]. All techniques have been implemented in the Zipperposition prover.

Hidden-Literal-Based Elimination For HLBE, an efficient representation of $\mathrm{HL}(L, N)$ is crucial. Because this set may be infinite, we underapproximate it by restricting the length of the transitive chains via a parameter K_{len}. Given the current clause set N, the finite map $\mathit{Imp}[L']$ associates with each literal L' a set of pairs (L, M) such that $L' \hookrightarrow^k L$, where $k \leq K_{\mathrm{len}}$ and M is the multiset of clauses used to derive $L' \hookrightarrow^k L$. Moreover, we consider only transitions of type 1 (as per Definition 8.4). The following algorithm maintains Imp dynamically, updating it as the prover derives and deletes clauses. It depends on the global variable Imp and the parameters K_{len} and K_{imp}.

procedure AddImplication($L_{\mathrm{a}}, L_{\mathrm{c}}, C$)
  if $\mathit{Imp}[\sigma(L_{\mathrm{a}})] \neq \emptyset$ for some renaming σ **then**
    $(L_{\mathrm{a}}, L_{\mathrm{c}}) \leftarrow (\sigma(L_{\mathrm{a}}), \sigma(L_{\mathrm{c}}))$
  if there are no L, L', M, σ such that $(L', M) \in \mathit{Imp}[L]$, $\sigma(L) = L_{\mathrm{a}}$, and $\sigma(L') = L_{\mathrm{c}}$ **then**
    for all (σ, M) such that $(\sigma(L_{\mathrm{c}}), M) \in \mathit{Imp}[\sigma(L_{\mathrm{a}})]$ **do**
      erase all (L', M') such that $M \subseteq M'$ from $\mathit{Imp}[\sigma(L_{\mathrm{a}})]$
    for all L such that $(L', M) \in \mathit{Imp}[L]$ and $\sigma(L_{\mathrm{a}}) = L'$ for some σ **do**
      if $|M| < K_{\mathrm{len}}$ **then**
        $\mathit{Imp}[L] \leftarrow \mathit{Imp}[L] \cup \{(\sigma(L_{\mathrm{c}}), M \uplus \{C\})\}$
    for all L such that $\mathit{Imp}[L] \neq \emptyset$ and $\sigma(L) = L_{\mathrm{c}}$ for some σ **do**
      $\mathit{Concl} \leftarrow \{(\sigma(L'), M \uplus \{C\}) \mid (L', M) \in \mathit{Imp}[L], |M| < K_{\mathrm{len}}\}$
      $\mathit{Imp}[L_{\mathrm{a}}] \leftarrow \mathit{Imp}[L_{\mathrm{a}}] \cup \mathit{Concl}$
    $\mathit{Congr} \leftarrow \{(s \not\approx t, \{C\}) \mid \exists u.\, L_{\mathrm{c}} = u[s] \not\approx u[t]\}$
    $\mathit{Imp}[L_{\mathrm{a}}] \leftarrow \mathit{Imp}[L_{\mathrm{a}}] \cup \{(L_{\mathrm{c}}, \{C\})\} \cup \mathit{Congr}$

15 **procedure** TrackClause(C)
  if $C = L_1 \vee L_2$ **then**
    AddImplication($\neg L_1$, L_2, C)
    AddImplication($\neg L_2$, L_1, C)
    if $L_2 = \sigma(\neg L_1)$ for some nonidempotent σ **then**
20   **for all** $i \leftarrow 1$ to K_{imp} **do**
        $L_2 \leftarrow \sigma(L_2)$
        AddImplication($\neg L_1$, L_2, C)

procedure UntrackClause(C)
  for all L_a, L_c, M such that $(L_c, M) \in \mathit{Imp}[L_a]$ **do**
25 **if** $C \in M$ **then**
      erase (L_c, M) from $\mathit{Imp}[L_a]$

The algorithm views a clause $L \vee L'$ as two implications $\neg L \rightarrow L'$ and $\neg L' \rightarrow L$. It stores only one entry for all literals equal up to variable renaming (line 2). Each implication $L_a \rightarrow L_c$ represented by the clause is stored only if its generalization is not present in *Imp* (line 4). Conversely, all instances of the implication are removed (line 5).

Next, the algorithm finds each implication stored in *Imp* that can be linked to $L_a \rightarrow L_c$: Either L_c becomes the new consequent (line 7) or L_a becomes the new antecedent (line 10). If L_c can be decomposed into $u[s] \not\approx u[t]$, rule 3 of Definition 8.4 allows us to store $s \not\approx t$ in $\mathit{Imp}[L_a]$ (line 14). This is an exception to the idea that transitive chains should use only rule 1. The application of rule 3 does not count toward the bound K_{len}. If L_a is of the form $u[s] \approx u[t]$, then *Imp* could be extended so that $\mathit{Imp}[s \approx t] = \mathit{Imp}[L_a]$, but this would substantially increase *Imp*'s memory footprint.

In first-order logic, different instances of the same clause can be used along a transitive chain. For example, the clause $C = \neg \mathsf{p}(x) \vee \mathsf{p}(\mathsf{f}(x))$ induces $\mathsf{p}(x) \hookrightarrow^i \mathsf{p}(\mathsf{f}^i(x))$ for all i. The algorithm discovers such self-implications (line 19): For each clause C of the form $\neg L \vee \sigma(L)$, where σ is some nonidempotent substitution, the pairs $(\sigma^2(L), \{C\}), \ldots, (\sigma^{K_{\text{imp}}+1}(L), \{C\})$ are added to $\mathit{Imp}[L]$, where K_{imp} is a parameter.

To track and untrack clauses efficiently, we implement the mapping *Imp* as a non-perfect discrimination tree [134]. Given a query literal L, this indexing data structure efficiently finds all literals L' such that for some σ, $\sigma(L') = L$ and $\mathit{Imp}[L'] \neq \emptyset$. We can use it to optimize all lookups except the one on line 7. For this remaining lookup, we add an index Imp^{-1} that inverts *Imp*, i.e., $\mathit{Imp}^{-1}[L] = \{L' \mid \mathit{Imp}[L'] = (L, M) \text{ for some } M\}$. To avoid sequentially going through all entries in *Imp* when the prover deletes them, for each clause C we keep track of each literal L such that C appears in $\mathit{Imp}[L]$. Finally, we limit the number of entries stored in $\mathit{Imp}[L]$ – by default, up to 48 pairs in each $\mathit{Imp}[L]$ are stored.

To implement the HLE rule, we use $\mathit{Imp}[L]$ as follows: Given a clause $C = L \vee L' \vee C'$, if there are two literals L_1, L_2 and a substitution σ such that $(L_2, M) \in \mathit{Imp}[L_1]$, $C \notin M$, $\sigma(L_1) = L$, and $\sigma(L_2) = L'$, we remove L from C. Literal L can also be removed if $\sigma(L_1) = \neg L'$ and $\sigma(L_2) = \neg L$. Rule HTR is implemented analogously.

The UnitHTR rule relies on maintaining the index *Unit*, which is built as follows. Whenever the prover derives a unit clause $C = \{L\}$, we find all entries L_a in *Imp* such that L_a and L are unifiable with the MGU σ. Then, we set $\mathit{Unit} \leftarrow \mathit{Unit} \cup \{(\sigma(L_c), M \cup \{C\}) \mid (L_c, M) \in \mathit{Imp}[L_a]\}$. Given a clause $L \vee C'$, we apply UnitHLE by looking for $(L', M) \in \mathit{Unit}$

such that $\sigma(L') = \neg L$, for some substitution σ; we apply UnitHTR by looking for L' such that $\sigma(L') = L$. The sets stored together with literals in *Unit* are used for building the proof object and to remove literals from *Unit* once a clause from the given set becomes redundant.

The same data structure is used for supporting FLE and FLR. When (L', M) is added to $Imp[L]$, we check whether $(\neg L', M') \in Imp[L]$ for some M'. If so, $(\neg L, M \cup M')$ is added to *Unit*.

In propositional logic, the conventional approach constructs the *binary implication graph* for the clause set N [77], with edges $(\neg L, L')$ and $(\neg L', L)$ whenever $L \vee L' \in N$. To avoid traversing the graph repeatedly, solvers rely on timestamps to discover connections between literals. This relies on syntactic literal comparisons, which is very fast in propositional logic but not in first-order logic, because of substitutions and congruence.

Predicate Elimination To implement portfolio predicate elimination, we maintain a record for each predicate symbol p occurring in the problem with the following fields: the set of definition clauses for p, the set of nondefinition clauses in which p occurs once, and the set of clauses in which p occurs more than once. These records are kept in a priority queue, prioritized by properties such as the presence of definition sets and the number of estimated resolutions. If p is the highest-priority symbol that is eligible for SPE or DPE, we eliminate it by removing all the clauses stored in p's record from the proof state and by adding flat resolvents to the passive set. Eliminating a symbol might make another symbol eligible.

As an optimization, predicate elimination keeps track only of symbols that appear at most K_{occ} times in the clause set. For inprocessing, we use signals that the prover emits whenever a clause is added to or removed from the proof state and update the records. At the beginning of the 1st, $(K_{\text{iter}} + 1)$st, $(2K_{\text{iter}} + 1)$st, … iteration of the given clause procedure's loop body, predicate elimination is systematically applied to the entire proof state. The first application of inprocessing amounts to preprocessing. After some informal experiments, we chose $K_{\text{occ}} = 512$ and $K_{\text{iter}} = 10$ as default values. The analogous optimization and limits apply for blocked clause elimination.

The most important novel aspect of our predicate elimination implementation is recognizing the definition clauses for a symbol p in a clause set N, which is performed as follows, assuming $\overline{x}$ is a fixed tuple of distinct free variables:

1. Let $G = \{C \mid C = (\neg)\mathsf{p}(\overline{y}) \vee C', C \in N$, no variable repeats in $\overline{y}$, and the variables of C' are all among $\overline{y}\}$. If G is empty, report failure; otherwise, continue.

2. Rename all clauses in G so that their only variables are $\overline{x}$.

3. Let $\lfloor a \rfloor$ be a function that assigns a propositional variable to each atom a. This function is lifted to literals by assigning $\lfloor \neg a \rfloor = \neg x$ if $\lfloor a \rfloor = x$, and to clauses pointwise. Furthermore, let $E = \{\lfloor C' \rfloor \mid (\neg)\mathsf{p}(\overline{x}) \vee C' \in G\}$. If E is satisfiable, report failure. Otherwise, let E' be an unsatisfiable core of E and G' the set of corresponding first-order clauses and continue.

4. If all resolvents in $G'_{\mathsf{p}} \bowtie_{\mathsf{p}} G'_{\neg\mathsf{p}}$ are tautologies, then G' is a definition set for symbol p. Otherwise, report failure.

The invalidity of set E from step 3 is checked using a SAT solver, which is already integrated in Zipperposition. As modern theorem provers (including E and Vampire) also use SAT solvers, this definition set recognition method can easily be implemented in those provers as well.

During experimentation, we noticed that recognizing definitions of symbols that occur in the conjecture often harms performance. Thus, Zipperposition recognizes definitions only for the remaining symbols.

8.7 Evaluation

We measure the impact of our elimination techniques for various values of their parameters. As a baseline, we use Zipperposition's first-order portfolio mode, which runs the prover in 13 configurations of heuristic parameters in consecutive time slices. None of these configurations use our new techniques. To evaluate a given parameter value, we fix it across all 13 configurations and compare the results with the baseline.

The benchmark set consists of all 13 495 CNF and FOF TPTP 7.3.0 theorems [157]. The experiments were carried out on StarExec servers [154] equipped with Intel Xeon E5-2609 CPUs clocked at 2.40 GHz. The portfolio mode uses a single CPU core with a CPU time limit of 180 s. The base configuration solves 7897 problems. The values in the tables indicate the number of problems solved minus 7897. Thus, positive numbers indicate gains over the baseline. The best result is shown in bold.

Hidden-Literal-Based Elimination The first experiments use all implemented HLBE rules. To avoid overburdening Zipperposition, we can enable an option to limit the number of tracked clauses for hidden literals. Once the limit has been reached, any request for tracking a clause will be rejected until a tracked clause is deleted. We can choose which kind of clauses are tracked: only clauses from the active set $\mathscr{A}$, only clauses from the passive set $\mathscr{P}$, or both. We also vary the maximal implication chain length K_{len} and the number of computed self-implications K_{imp}.

In Zipperposition, every lookup for instances or generalizations of $s \approx t$ must be done once for each orientation of the equation. To avoid this inefficiency, and also because the implementation of hidden literals does not fully exploit congruence, we can disable tracking clauses with at least one functional literal. Clauses containing functional literals can then still be simplified.

Figures 8.1 and 8.2 show the results, without and with functional literal tracking enabled, for $K_{\text{len}} = 2$ and $K_{\text{imp}} = 0$. The columns specify different limits on the number of tracked clauses, with ∞ denoting that no limit is imposed. The rows represent different kinds of tracked clauses. The results suggest that tracking functional literals is not worth the effort but that tracking predicate literals is. The best improvement is observed when both active and passive clauses are tracked. Normally DISCOUNT-loop provers [4] such as Zipperposition do not simplify active clauses using passive clauses, but here we see that this can be effective. Figure 8.3 shows the impact of varying K_{len} and K_{imp}, when 500 clauses from the entire proof state are tracked. These results suggest that computing long implication chains is counterproductive.

	Tracked clauses			
	250	500	1000	∞
Active	−14	−16	−8	−12
Passive	+7	+10	+5	−35
Both	**+12**	+10	+7	−45

Figure 8.1: Impact of the number and kinds of tracked clauses on HLBE performance, when only predicate literals are tracked

	Tracked clauses			
	250	500	1000	∞
Active	−10	−14	−8	−18
Passive	−5	−5	−14	−71
Both	**+2**	−1	−8	−79

Figure 8.2: Impact of the number and kinds of tracked clauses on HLBE performance, when all literals are tracked

	Chain length K_{len}			
	1	2	4	8
$K_{\text{imp}} = 0$	+9	+10	+7	+5
$K_{\text{imp}} = 1$	+5	**+11**	+7	+4
$K_{\text{imp}} = 2$	+6	**+11**	+8	+8

Figure 8.3: Impact of the parameters K_{len} and K_{imp} on HLBE performance

		Relaxed with K_{tol}				
	K&K	0	25	50	100	200
SPE preprocessing	+70	+117	+154	+160	+154	+158
PPE preprocessing	+71	+124	+160	+164	**+165**	+162

Figure 8.4: Impact of the choice of criterion on predicate elimination performance

Predicate and Blocked Clause Elimination For defined predicate elimination, the number of resolvents grows exponentially with the number of occurrences of p. To avoid this expensive computation, we limit the applicability of PPE to proof states for which p is singular. According to our informal experiments, full PPE, without this restriction, generally performs less well.

Predicate elimination can be done using Khasidashvili and Korovin's criterion (K&K) or using our relaxed criterion with different values of K_{tol}. Figure 8.4 shows the results for SPE and PPE used as preprocessors. Our numbers corroborate Khasidashvili and Korovin's findings: SPE with K&K proves 70 more problems than the base, a 0.9% increase, comparable to the 1.8% they observe when they combine SPE with additional preprocessing. Remarkably, the number of additional proved problems more than doubles when we use our criterion with $K_{\text{tol}} > 0$, for both SPE and PPE.

Although this is not evident in Figure 8.4, varying K_{tol} substantially changes the set of problems solved. For example, when $K_{\text{tol}} = 0$, SPE proves 60 theorems not proved using $K_{\text{tol}} = 50$. The effect weakens as K_{tol} grows. When $K_{\text{tol}} = 100$, SPE proves only 13 problems not found when $K_{\text{tol}} = 200$. Similarly, the set of problems proved by SPE and PPE differs: When $K_{\text{tol}} = 25$, 14 problems are proved by PPE but missed by SPE. Recognizing definition sets is useful: PPE outperforms SPE regardless of the criterion.

Performing BCE and variable elimination until fixpoint increases the performance of

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	+30	+154	+159	+160	+**166**	+162
Inprocessing	−48	+140	+127	+146	+131	+127

Figure 8.5: Performance of predicate and blocked clause elimination

	BCE	SPE	SPE +BCE	PPE	PPE +BCE	HLBE +PPE +BCE
Preprocessing	+29	+46	+**60**	+47	+59	+55

Figure 8.6: Performance of predicate and blocked clause elimination for establishing satisfiability

SAT solvers [85]. We can check whether the same holds for superposition provers. In this experiment, we use the relaxed criterion with $K_{\text{tol}} = 25$ and HLBE which tracks up to 500 clauses from any clause set, $K_{\text{len}} = 2$, and $K_{\text{imp}} = 0$. We use each technique as preprocessing and inprocessing.

The results are summarized in Figure 8.5, where the + sign denotes the combination of techniques. We confirm the results obtained by Kiesl et al. about the performance of BCE as preprocessing: It helps prove 30 more problems from our benchmark set, increasing the success rate by roughly 0.4%. The same percentage increase was obtained by Kiesl et al. Using BCE as inprocessing, however, hurts performance, presumably because of its incompatibility with the redundancy criterion.

For preprocessing, the combinations SPE+BCE and PPE+BCE performed roughly on a par with SPE and PPE, respectively. This stands in contrast to the situation with SAT solvers, where such a combination usually helps. It is also worth noting that the inprocessing techniques never outperform their preprocessing counterparts. The last column shows that combining HLBE with other elimination techniques overburdens the prover.

Satisfiability by Blocked Clause Elimination Kiesl et al. found that blocked clause elimination is especially effective on satisfiable problems. To corroborate their results and ascertain whether a combination of predicate elimination and blocked clause elimination increases the success rate, we evaluate BCE on all 2273 satisfiable TPTP FOF and CNF problems. The hardware and CPU time limits are the same as in the experiments above. Figure 8.6 presents the results.

The baseline establishes the satisfiability of 856 problems. We consider only preprocessing techniques, since BCE compromises refutational completeness—a saturation does not guarantee that the original problem is satisfiable. We note that recognizing definition sets makes almost no difference on satisfiable problems. The sets of problems solved by BCE and PPE differ—30 problems are solved by BCE and not by PPE.

8.8 Discussion and Related Work

We briefly surveyed related work in Sect. 8.1. In this section, we give a more detailed overview and further discuss connections with related work.

The research presented in this chapter is two-pronged. For SAT elimination techniques already generalized to preprocess first-order problems, we looked for ways to interleave them with the given clause procedure of a superposition prover, as inprocessing. For techniques that had not yet been ported to first-order logic, we looked for generalizations that allow both preprocessing and inprocessing.

Hidden tautology elimination was first described by Heule et al. [76]. A better implementation that also supports hidden literal elimination was later described by the same group of authors [77]. We generalized the underlying theoretical concepts to first-order logic, and provided an efficient way to deal with the infinite number of hidden literals that arise with this generalization. More efficient graph-based techniques are yet to be explored.

Variable elimination, based on Davis–Putnam resolution [52], has been studied in the context of both propositional logic [46, 155] and QBF [26]. It was generalized to first-order logic (as a preprocessor) by Khasidashvili and Korovin [91], yielding a technique called predicate elimination. An improvement of variable elimination, that uses formula definition information, has been popularized as a preprocessing and inprocessing technique for CDCL solvers by Eén and Biere [59]. We generalized this improvement to first-order logic and combined it with Khasidashvili and Korovin's approach. With tolerable restrictions, this extension can be used as an inprocessing technique. In SAT and QBF, it was observed that allowing variable elimination to slightly increase the clause set size improves performance [27]. We implemented a similar approach, achieving double the number of additional proofs found compared to more restrictive approaches.

Blocked clause elimination is used in both SAT [85] and QBF solvers [27]. Its generalization to first-order logic [93] has showed positive effects when used as a preprocessor. We showed that blocked clauses cannot be removed during saturation, but that they can be effectively used to show satisfiability of the clause set. A combination of blocked clause elimination and variable elimination performs well in propositional logic [85], but we observed no comparable improvement when their generalizations are combined.

Our general approach is one of many ways to combine ideas from SAT solving and first-order proving. Other noteworthy architectures that either incorporate a SAT solver or that generalize the CDCL calculus include DPLL(T) with quantifier instantiation [10, 121, 138], DPLL($\Gamma + T$) [36], labeled splitting [63], AVATAR [167], MCSAT [120], CDSAT [35], and SGGS [37].

8.9 Conclusion

We adapted several preprocessing and inprocessing elimination techniques implemented in modern SAT solvers so that they work in a superposition prover. This involved lifting the techniques to first-order logic with equality but also tailoring them to work in tandem with superposition and its redundancy criterion. Although SAT solvers and superposition provers embody radically different philosophies, we found that the lifted SAT techniques provide valuable optimizations.

We see several avenues for future work. First, the implementation of hidden literals could be extended to exploit equality congruence. Second, although blocked clause elimination is generally incomplete as an inprocessing technique, we hope to achieve refutational completeness for a substantial fragment of it. Third, predicate and blocked clause elimination, which thrive on the absence of clauses from the proof state, could be enhanced by tagging and ignoring generated clauses that have not yet been used to subsume or simplify untagged clauses. Fourth, predicate and blocked clause elimination could be extended to work with functional literals. Fifth, more SAT techniques could be adapted, including bounded variable addition [111] and blocked clause addition [101]. Sixth, the techniques we covered could be adapted to work with other first-order calculi, or generalized further to work with higher-order calculi such as combinatory superposition [25] and $o\lambda$Sup.

9

Conclusion and Future Work

As this thesis is brought to an end, I would like to use these last pages to reflect not only on the work I did in the last four years, but also on the many bumps on the road that I did not get to describe in the preceding chapters. When it comes to reflections, the performance art piece Rhythm 10, performed by Marina Abramović, one of the greatest Yugoslav artists, comes to my mind. The themes that it explores such as taking risks and the inevitability of making mistakes were the root of fears and doubts that I was slowly overcoming over the last four years. The photograph on the cover of this thesis was taken when it was first performed, in Edinburgh, in 1973. The instructions for the performance read:

> Preparation
>
> I place a sheet of white paper on the floor.
> I lay 20 knives of different sizes and shapes on the floor.
> I place 2 tape recorders with microphones on the floor.
>
> Performance
>
> I turn on the first tape recorder.
> I take the first knife and stab in between the fingers of my left hand as fast as possible.
> Every time I cut myself, I change to a different knife.
> When I've used all of the knives (all the rhythms), I rewind the tape recorder.
> I listen to the recording of the first part of the performance.
> I concentrate.
> I repeat the first part of the performance.
> I take the knives in the same order, follow to the same order, follow to the same rhythm and cut myself in the same places.
> In this performance, the mistakes of time past and time present are synchronized.
> I rewind the second tape recorder and listen to the double rhythm of the knives.
> I leave.

The work described in this thesis was cyclic: It consists of three phases (three stops as described in Chapter 1), each of which has the same parts: theory (calculus and algorithms), engineering (implementation), and optimization (heuristics). While the calculi were mostly developed by Alexander Bentkamp [15, 17, 18], the task of developing algorithms for indexing or unification, as well as the tasks of implementing and optimizing the implementation, were mainly mine.

Structuring the project cyclically turned out to be a great idea of Jasmin Blanchette. When I first started doing the research which would later form the basis of Chapter 3, I was overwhelmed by the scope of the project and felt uncertain if E could ever fully support λ-free higher-order logic. Many of the extensions described in the later chapters were even larger in scope. However, as each extension relied on the previous one, I quickly developed a feeling of where the possible pitfalls are and how to avoid them. This gave me an invaluable source of encouragement, without which I could hardly have arrived at the end of the project.

The work in Chapter 3 completed the first cycle. Soon after the original paper introducing Ehoh [169] was published, Ehoh found its use as the backend of Sledgehammer and Satallax, showing that even when limited to a fragment of higher-order logic, efficient higher-order reasoning is in demand.

The goal of the next cycle was to support λ-abstraction. When we first implemented λSup, the calculus that achieves this goal, we quickly realized that one of its main flaws was that it relies on enumerating full unifiers. In comparison, some older resolution-based higher-order calculi enumerate preunifiers, which are less explosive and thus more manageable. This is why we focused on developing a full unification procedure that removes the redundancy present in other procedures. It also implements many advancements and optimizations described in the unification literature after the introduction of the most influential full unification procedure, Jensen-Pietrzykowski's procedure in the 1970s. We also made sure that our procedure can easily be customized to trade bits of its completeness for substantially improved performance. This procedure, described in Chapter 4, transformed the λSup calculus into a competitive higher-order calculus.

Even though λSup became competitive, its implementation in Zipperposition lagged behind the competition. To investigate why this is the case, we manually analyzed hundreds of benchmarks which the competition proves, but on which Zipperposition failed. This analysis forms the basis of work that is described in Chapters 5 and 6. The main conclusion was that the competition, most notably the tableaux-based Satallax, reasons better with formulas. Tableaux inferences are more intuitive than the ones that superposition or resolution provers perform and are more in line with how a mathematician structures a proof. Fitting these kind of inferences into the superposition context was challenging, but it proved very successful. After they were implemented in Zipperposition, it took its first victory at the higher-order division of CASC.

After its first victory, Zipperposition was successfully integrated into Sledgehammer and it is now part of Sledgehammer's default installation. We also noticed that problems which only Zipperposition can solve do pop up from time to time in practice. Antoine Defourné also integrated Zipperposition in the TLA+ proof assistant and observed that it helps prove some problems that were previously out of reach [53].

Our extension of λSup was guided by performance, rather than completeness with

respect to full higher-order logic. After the first victory, we were left wondering what was the kind of problems that our implementation cannot solve, and started looking for a complete full higher-order calculus. At the end, we designed *oλ*Sup, which gave us the precise answer in the form of problems that cannot be solved and the rules which are necessary to solve them.

While preparing for the following year's CASC competition, we found problems that occur in practice for which the rules of *oλ*Sup are necessary. However, these problems occur in less than one percent of the whole TPTP library. Furthermore, some of *oλ*Sup's rules are so explosive that they are disabled in most CASC portfolio configurations.

This led us to take a radically different approach when extending Ehoh to λE. We envisioned λE as a prover that excels on problems coming from proof assistants. Unlike in TPTP, on these benchmarks hard, hand-crafted mathematical puzzles rarely occur. Thus, we conjectured that extending λfSup with the most rudimentary features of λ-superposition such as full higher-order unification and β-reduction-aware term orders would help us prove most of higher-order problems without the explosion of complete approaches.

While this proved mostly true, we quickly realized that λE should assimilate most of the incomplete Boolean reasoning techniques described in Chapters 5 and 6. All of them were straightforward to implement except for dynamic clausification, which we did not implement. It is possible that support for this technique does not require profound changes to E(hoh)'s formula treatment, but due to a lack of time we did not explore this alley.

This highly pragmatic approach proved successful not only on proof assistant benchmarks: λE also excels on TPTP benchmarks, trailing slightly behind Zipperposition. In the coming months we plan to replace Ehoh by λE in Sledgehammer. As Ehoh is already one of the most successful Sledgehammer backends, this will further improve the efficiency of hundreds of mathematicians and computer scientists using Isabelle. By manually inspecting the TPTP benchmarks that are out of reach for λE, but within the reach of Zipperposition, we realized that resolving the lack of dynamic clausification might be the weight necessary to tip the scales in favor of λE on TPTP benchmarks.

This extension of Ehoh to λE finished the third cycle. Inspired by our approach of extending techniques designed for weaker logics to work in the context of richer logics, we decided to look for ways in which superposition can assimilate the most successful techniques of SAT solving. Extension of hidden literals was straightforward, but finding the right way to tame their infinite nature in the first-order case, as well as the right way to integrate equality in their definition, was challenging. Predicate elimination posed challenges in terms of finding the exact conditions under which it can be integrated with the saturation loop. We first observed that blocked clause elimination destroys the completeness of superposition calculus when we noticed that Zipperposition does not prove some problems when the technique is enabled. After sifting through tens of benchmarks and carefully examining the debugging information, we realized that this was not due to a bug in the implementation but due to the incompatibility of blocked clause elimination with the redundancy criterion. In the end, we were left somewhat disappointed to learn that SAT techniques do not scale well enough to be used as superposition simplification techniques. However, we learned that they are very successful as preprocessing techniques.

Future Work

Even though the work in this thesis completes the three-step cycle, there are still many unexplored alleys for future work. We separate them by the topic explored in one of the earlier chapters:

Higher-Order Unification The complete variant of the unification procedure described in Chapter 4 is proved complete with substantial restrictions on the order and kind of applied rules. To further remove redundancy, we hipe to find alternative, less explosive unification rules that do not compromise completeness. Furthermore, we want to investigate if there are more combinations of unification rules that lead to redundant unifiers and find ways to remove those redundancy-inducing combinations.

We also described only one pragmatic variant of the unification procedure. This variant was designed after we sifted through hundreds of benchmarks to get a taste of what unifiers occur in practice. It is possible that this pragmatic variant can be further tweaked to perform more efficiently without loosing many useful unifiers.

Lastly, we fixed the order in which Zipperposition applies different unification rules. It would be interesting to see if applying imitation before projection (or vice versa) yields performance improvements on some kinds of problems. We also postponed the application of the most explosive rules such as iteration to the very end. However, it would not be surprising if there are many hard hand-crafted problems on which applying these rules early on would lead to a proof more quickly.

Pragmatic Techniques and Heuristics Many of the Boolean reasoning techniques described in Chapter 5 were either inspired by the ones implemented in traditional higher-order theorem provers or by the unsolved TPTP benchmarks. Since the work present in Chapter 5 anticipated the complete higher-order calculus that could make these techniques obsolete, we implemented only the most successful ones that are easy to port to saturation-based provers. Most notably, unlike Satallax [40], Zipperposition does not use the fact that there only finitely many functions of a type built using only the function type constructor ($\rightarrow$) and Boolean type (o). Implementing techniques for efficient enumeration of all Boolean functions of a given type could help Zipperposition prove problems that are still out of its reach, but can be solved by Satallax.

In Chapter 6 we explored how we keep the explosion inherent to the $o\lambda$Sup calculus under control. Due to time and resource constraints, in the evaluation of our new given-clause loop we fixed some parameters that control which streams are queried. We plan to tune these parameters in the future. Similarly, we provided only some of the clause priority functions that rely on higher-order features. We plan to investigate more proofs in detail to create new priority functions that more cautiously separate less and more useful higher-order inferences. Lastly, we observed that use of Ehoh as a backend greatly improves the performance of Zipperposition. We plan to further investigate the proofs that Ehoh returns and find out how Zipperposition can find them without the use of a backend.

Further Extensions of λE λE outperforms other provers on problems coming from proof assistants. However, its performance on TPTP problems is slightly behind Zipper-

position. We plan to further investigate the benchmarks on which λE fails, but that are proved by Zipperposition. One of the first features we plan to implement is dynamic clausification, which can possibly help improve λE's performance on first-order problems as well. We also plan to experiment with implementing some of the *oλ*Sup rules necessary for completeness such as FluidSup. Heuristics for λE are trained on a subset of Sledgehammer and TPTP problems. We plan to use other benchmark sets such as GRUNGE [41] to make λE efficient on a wider range of higher-order benchmarks.

SAT-Inspired Eliminations for Superposition We already implemented some of the techniques described in Chapter 8 in E. However, this implementation is not properly tested and optimized. We plan to put finishing touches to this implementation and test the effects of implemented preprocessing techniques on E's performance. There are more techniques used in SAT solving, notably bounded variable addition [111] and blocked clause addition [101], that we did not port to first-order logic. We want to investigate whether these techniques can be lifted to first-order logic. Lastly, we already started work on porting existing techniques to higher-order logic.

Bibliography

References

[1] Andrews, P.B.: Resolution in type theory. J. Symb. Log. 36(3), 414–432 (1971)

[2] Andrews, P.B.: Classical type theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 965–1007. Elsevier and MIT Press (2001)

[3] Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: A theorem-proving system for classical type theory. J. Autom. Reason. 16(3), 321–353 (1996)

[4] Avenhaus, J., Denzinger, J., Fuchs, M.: DISCOUNT: A system for distributed equational deduction. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914, pp. 397–402. Springer (1995)

[5] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)

[6] Bachmair, L., Ganzinger, H.: Non-clausal resolution and superposition with selection and redundancy criteria. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 273–284. Springer (1992)

[7] Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. 4(3), 217–247 (1994)

[8] Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 19–99. Elsevier and MIT Press (2001)

[9] Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. J. Autom. Reason. 47(4), 451–479 (2011)

[10] 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 (2017)

[11] Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE-27. LNCS, vol. 11716, pp. 35–54. Springer (2019)

[12] 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)

[13] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 432–453. Springer (2017)

[14] Beeson, M.: Lambda logic. In: Basin, D.A., Rusinowitch, M. (eds.) IJCAR 2004. LNCS, vol. 3097, pp. 460–474. Springer (2004)

[15] Bentkamp, A., Blanchette, J., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. Log. Methods Comput. Sci. 17(2) (2021)

[16] Bentkamp, A., Blanchette, J., Nummelin, V., Tourret, S., Vukmirović, P., Waldmann, U.: Mechanical mathematicians (2022), `https://matryoshka-project.github.io/pubs/mechanical.pdf`, draft article

[17] Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, vol. 12699, pp. 396–412. Springer (2021)

[18] Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. J. Autom. Reason. 65(7), 893–940 (2021)

[19] Benzmüller, C., Kohlhase, M.: System description: LEO—a higher-order theorem prover. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 139–144. Springer (1998)

[20] Benzmüller, C., Miller, D.: Automation of higher-order logic. In: Siekmann, J.H. (ed.) Computational Logic, Handbook of the History of Logic, vol. 9, pp. 215–254. Elsevier (2014)

[21] Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Can a higher-order and a first-order theorem prover cooperate? In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 415–431. Springer (2004)

[22] Benzmüller, C., Sultana, N., Paulson, L.C., Theiss, F.: The higher-order prover Leo-II. J. Autom. Reason. 55(4), 389–404 (2015)

[23] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)

[24] Bhayat, A., Reger, G.: Restricted combinatory unification. In: Fontaine, P. (ed.) CADE-27. LNCS, vol. 11716, pp. 74–93. Springer (2019)

[25] Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 278–296. Springer (2020)

[26] Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer (2004)

[27] Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNCS, vol. 6803, pp. 101–115. Springer (2011)

[28] Blanchette, J., Fontaine, P., Schulz, S., Tourret, S., Waldmann, U.: Stronger higher-order automation: A report on the ongoing matryoshka project. In: Suda, M., Winkler, S. (eds.) EPTCS 311: Proceedings of the Second International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements - Natal, Brazil, August 26, 2019. pp. 11–18. Electronic Proceedings in Theoretical Computer Science, EPTCS, EPTCS (12 2019)

[29] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Log. Meth. Comput. Sci. 12(4), 13:1–13:52 (2016)

[30] Blanchette, J.C., Greenaway, D., Kaliszyk, C., Kühlwein, D., Urban, J.: A learning-based fact selector for Isabelle/HOL. J. Autom. Reason. 57(3), 219–244 (2016)

[31] Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE. LNCS, vol. 7898, pp. 414–420. Springer (2013)

[32] Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 461–479. Springer (2017)

[33] Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wroclaw, Poland (2011)

[34] Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)

[35] Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Satisfiability modulo theories and assignments. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 42–59. Springer (2017)

[36] Bonacina, M.P., Lynch, C., de Moura, L.: On deciding satisfiability by DPLL($\Gamma + T$) and unsound theorem proving. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 35–50. Springer (2009)

[37] Bonacina, M.P., Plaisted, D.A.: SGGS theorem proving: An exposition. In: Schulz, S., de Moura, L., Konev, B. (eds.) PAAR-2014. EPiC Series in Computing, vol. 31, pp. 25–38. EasyChair (2014)

[38] Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 151–156. Springer (2009)

[39] Brown, C.E.: Satallax: An automatic higher-order prover. In: IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer (2012)

[40] Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. J. Autom. Reason. 51(1), 57–77 (2013)

[41] Brown, C.E., Gauthier, T., Kaliszyk, C., Sutcliffe, G., Urban, J.: GRUNGE: A grand unified ATP challenge. In: Fontaine, P. (ed.) CADE-27. LNCS, vol. 11716, pp. 123–141. Springer (2019)

[42] Bruijn, N.G.D.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. J. Symb. Log. 40(3), 470–470 (1975)

[43] Carnielli, W., Coniglio, M.E., Marcos, J.: Logics of formal inconsistency. In: Gabbay, D.M., Guenthner, F. (eds.) Handbook of Philosophical Logic. Handbook of Philosophical Logic, vol. 14, pp. 1–93. Springer (2007)

[44] Chang, C., Lee, R.C.T.: Symbolic logic and mechanical theorem proving. Computer science classics, Academic Press (1973)

[45] Charguéraud, A.: The locally nameless representation. J. Autom. Reason. 49(3), 363–408 (2012)

[46] Chatalic, P., Simon, L.: ZRES: The old Davis–Putnam procedure meets ZBDD. In: McAllester, D.A. (ed.) CADE-18. LNCS, vol. 1831, pp. 449–454. Springer (2000)

[47] Church, A.: A note on the entscheidungsproblem. J. Symb. Log. 1(1), 40–41 (1936)

[48] Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. (Extensions de la Superposition pour l'Arithmétique Linéaire Entière, l'Induction Structurelle, et bien plus encore). Ph.D. thesis, École Polytechnique, Palaiseau, France (2015)

[49] Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS, vol. 10483, pp. 172–188. Springer (2017)

[50] Czajka, Ł.: Improving automation in interactive theorem provers by efficient encoding of lambda-abstractions. In: Avigad, J., Chlipala, A. (eds.) CPP 2016. pp. 49–57. ACM (2016)

[51] Czajka, L., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory. J. Autom. Reason. 61(1-4), 423–453 (2018)

[52] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)

[53] Defourné, A.: Improving automation for higher-order proof steps. In: Konev, B., Reger, G. (eds.) FroCoS. LNCS, vol. 12941, pp. 139–153. Springer (2021)

[54] Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM 22(8), 465–476 (1979)

[55] Desharnais, M., Vukmirović, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer, https://matryoshka-project.github.io/pubs/seventeen.pdf, draft paper

[56] Dougherty, D.J.: Higher-order unification via combinators. Theor. Comput. Sci. 114(2), 273–298 (1993)

[57] Dowek, G.: Higher-order unification and matching. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1009–1062. Elsevier and MIT Press (2001)

[58] Ebner, G., Blanchette, J., Tourret, S.: Unifying splitting. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, vol. 12699, pp. 344–360. Springer (2021)

[59] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer (2005)

[60] Färber, M., Brown, C.E.: Internal guidance for Satallax. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 349–361. Springer (2016)

[61] Farmer, W.M.: A unification algorithm for second-order monadic terms. Ann. Pure Appl. Logic 39(2), 131–174 (1988)

[62] Ferreirós, J.: The road to modern logic - an interpretation. Bull. Symb. Log. 7(4), 441–484 (2001)

[63] Fietzke, A., Weidenbach, C.: Labelled splitting. Ann. Math. Artif. Intell. 55(1–2), 3–34 (2009)

[64] Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer (2013)

[65] Fitting, M.: First-Order Logic and Automated Theorem Proving. Graduate Texts in Computer Science, Springer, 2nd edn. (1996)

[66] Freeman, J.W.: Improvements to Propositional Satisfiability Search Algorithms. Ph.D. thesis, University of Pennsylvania (1995)

[67] Gabbay, D.M., Ohlbach, H.J.: Quantifier elimination in second-order predicate logic. In: Nebel, B., Rich, C., Swartout, W.R. (eds.) KR '92. pp. 425–435. Morgan Kaufmann (1992)

[68] Gallier, J.H.: Logic for Computer Science: Foundations of Automatic Theorem Proving. Wiley (1987)

[69] Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. Inf. Comput. 199(1-2), 3–23 (2005)

[70] Gleiss, B., Suda, M.: Layered clause selection for theory reasoning (short paper). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020, Part I. LNCS, vol. 12166, pp. 402–409. Springer (2020)

[71] Gonthier, G.: Formal proof—The four-color theorem. Notices of the AMS 55(11), 1382–1393 (2008)

[72] Gödel, K.: Die vollständigkeit der axiome des logischen funktionenkalküls. Monatshefte für Mathematik und Physik 37, 349–360 (1930)

[73] Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J.M., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture. CoRR abs/1501.02155 (2015)

[74] Harrison, J.: HOL light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs. LNCS, vol. 5674, pp. 60–66. Springer (2009)

[75] Heule, M., Seidl, M., Biere, A.: A unified proof system for QBF preprocessing. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 91–106. Springer (2014)

[76] Heule, M.J.H., Järvisalo, M., Biere, A.: Clause elimination procedures for CNF formulas. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 357–371. Springer (2010)

[77] Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 201–215. Springer (2011)

[78] Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 313–329. Springer (2016)

[79] Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: Mertsching, B., Hund, M., Aziz, M.Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 435–443. Springer (2009)

[80] Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNCS, vol. 6803, pp. 299–314. Springer (2011)

[81] Huet, G.P.: A mechanization of type theory. In: Nilsson, N.J. (ed.) IJCAI-73. pp. 139–146. William Kaufmann (1973)

[82] Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. 1(1), 27–57 (1975)

[83] Hughes, R.J.M.: Super combinators: A new implementation method for applicative languages. In: Park, D.M.R., Friedman, D.P., Wise, D.S., Jr., G.L.S. (eds.) Symposium on LISP and Functional Programming. pp. 1–10. ACM (1982)

[84] Huth, M., Ryan, M.D.: Logic in Computer Science - Modelling and Reasoning about Systems. Cambridge University Press (2000)

[85] Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 129–144. Springer (2010)

[86] Jensen, D.C., Pietrzykowski, T.: Mechanizing omega-order type theory through unification. Theor. Comput. Sci. 3(2), 123–171 (1976)

[87] Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer (1985)

[88] Kaliszyk, C., Sutcliffe, G., Rabe, F.: TH1: the TPTP typed higher-order form with rank-1 polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) PAAR@IJCAR. CEUR Workshop Proceedings, vol. 1635, pp. 41–55. CEUR-WS.org (2016)

[89] Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Math. Comput. Sci. 9(1), 5–22 (2015)

[90] Kamareddine, F.: Reviewing the classical and the De Bruijn notation for lambda-calculus and pure type systems. J. Log. Comput. 11(3), 363–394 (2001)

[91] Khasidashvili, Z., Korovin, K.: Predicate elimination for preprocessing in first-order theorem proving. In: Creignou, N., Berre, D.L. (eds.) SAT 2016. LNCS, vol. 9710, pp. 361–372. Springer (2016)

[92] Kiesl, B., Suda, M.: A unifying principle for clause elimination in first-order logic. In: de Moura, L. (ed.) CADE-26. LNCS, vol. 10395, pp. 274–290. Springer (2017)

[93] Kiesl, B., Suda, M., Seidl, M., Tompits, H., Biere, A.: Blocked clauses in first-order logic. In: Eiter, T., Sands, D. (eds.) LPAR-21. EPiC Series in Computing, vol. 46, pp. 31–48. EasyChair (2017)

[94] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) SOSP 2009. pp. 207–220. ACM (2009)

[95] 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 (1970)

[96] Kohlhase, M.: A mechanization of sorted higher-order logic based on the resolution principle. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (1994)

[97] Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 292–298. Springer (2008)

[98] Kotelnikov, E., Kovács, L., Suda, M., Voronkov, A.: A clausal normal form translation for FOOL. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. EPiC Series in Computing, vol. 41, pp. 53–71. EasyChair (2016)

[99] Kotelnikov, E., Kovács, L., Voronkov, A.: A first class Boolean sort in first-order theorem proving and TPTP. In: CICM 2015. pp. 71–86 (2015)

[100] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer (2013)

[101] Kullmann, O.: On a generalization of extended resolution. Discr. Appl. Math. 96–97, 149–176 (1999)

[102] Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer (2010)

[103] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)

[104] Levecque, K., Anseel, F., De Beuckelaer, A., Van der Heyden, J., Gisle, L.: Work organization and mental health problems in PhD students. Research Policy 46(4), 868–879 (2017)

[105] Libal, T., Miller, D.: Functions-as-constructors higher-order unification. In: Kesner, D., Pientka, B. (eds.) FSCD 2016. LIPIcs, vol. 52, pp. 26:1–26:17. Schloss Dagstuhl (2016)

[106] Libal, T., Steen, A.: Towards a substitution tree based index for higher-order resolution theorem provers. In: Fontaine, P., Schulz, S., Urban, J. (eds.) PAAR 2016. CEUR-WS, vol. 1635, pp. 82–94. CEUR-WS (2016)

[107] Lindblad, F.: A focused sequent calculus for higher-order logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 61–75. Springer (2014)

[108] Löchner, B.: Things to know when implementing KBO. J. Autom. Reason. 36(4), 289–310 (2006)

[109] Löchner, B., Schulz, S.: An evaluation of shared rewriting. In: de Nivelle, H., Schulz, S. (eds.) IWIL-2001. pp. 33–48. Max-Planck-Institut für Informatik (2001)

[110] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. In: Buchanan, B.G. (ed.) IJCAI-79. pp. 542–551. William Kaufmann (1979)

[111] Manthey, N., Heule, M., Biere, A.: Automated reencoding of Boolean formulas. In: Biere, A., Nahir, A., Vos, T.E.J. (eds.) HVC 2012. LNCS, vol. 7857, pp. 102–117. Springer (2012)

[112] Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009)

[113] McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reason. 9(2), 147–167 (1992)

[114] McCune, W.: Solution of the Robbins problem. J. Autom. Reason. 19(3), 263–276 (1997)

[115] McCune, W.: OTTER 3.3 reference manual. CoRR cs.SC/0310056 (2003)

[116] McCune, W., Wos, L.: Otter—the CADE-13 competition incarnations. J. Autom. Reason. 18(2), 211–220 (1997)

[117] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. 40(1), 35–60 (2008)

[118] Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press (2012)

[119] Miller, D.A.: A compact representation of proofs. Stud. Log. 46(4), 347–370 (1987)

[120] de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer (2013)

[121] de Moura, L.M., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE-21. LNCS, vol. 4603, pp. 183–198. Springer (2007)

[122] Murray, N.V.: Completely non-clausal theorem proving. Artif. Intell. 18(1), 67–85 (1982)

[123] Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)

[124] Nipkow, T.: Functional unification of higher-order patterns. In: Best, E. (ed.) LICS 1993. pp. 64–74. IEEE Computer Society (1993)

[125] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[126] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 335–367. Elsevier and MIT Press (2001)

[127] Nummelin, V., Bentkamp, A., Tourret, S., Vukmirović, P.: Superposition with first-class Booleans and inprocessing clausification. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, Springer (2021)

[128] Ohlbach, H.J.: SCAN—elimination of predicate quantifiers. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE-13. LNCS, vol. 1104, pp. 161–165. Springer (1996)

[129] Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999)

[130] Paulson, L.C.: Isabelle: The next seven hundred theorem provers. In: Lusk, E.L., Overbeek, R.A. (eds.) CADE-9. LNCS, vol. 310, pp. 772–773. Springer (1988)

[131] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)

[132] Peltier, N.: A variant of the superposition calculus. Archive of Formal Proofs (2016), https://www.isa-afp.org/

[133] Prehofer, C.: Solving higher order equations: from logic to programming. Ph.D. thesis, Technical University Munich, Germany (1995)

[134] Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Handbook of Automated Reasoning, vol. 2, pp. 1853–1964. Elsevier and MIT Press (2001)

[135] Reger, G., Suda, M.: Checkable proofs for first-order theorem proving. In: Reger, G., Traytel, D. (eds.) ARCADE 2017. EPiC Series in Computing, vol. 51, pp. 55–63. EasyChair (2017)

[136] Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNCS, vol. 9195, pp. 399–415. Springer (2015)

[137] Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016)

[138] Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part II. LNCS, vol. 10806, pp. 112–131. Springer (2018)

[139] Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Nebel, B. (ed.) IJCAI 2001. pp. 611–617. Morgan Kaufmann (2001)

[140] Robinson, J.: Mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 151–170. Edinburgh University Press (1969)

[141] Robinson, J.: A note on mechanizing higher order logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 5, pp. 121–135. Edinburgh University Press (1970)

[142] Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (1965)

[143] Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)

[144] Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 477–483. Springer (2012)

[145] Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics—Essays in Memory of William W. McCune. LNCS, vol. 7788, pp. 45–67. Springer (2013)

[146] Schulz, S.: We know (nearly) nothing! But can we learn? In: Reger, G., Traytel, D. (eds.) ARCADE 2017. EPiC Series in Computing, vol. 51, pp. 29–32. EasyChair (2017)

[147] Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE-27. LNCS, vol. 11716, pp. 495–507. Springer (2019)

[148] Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 330–345. Springer (2016)

[149] Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD 1996. pp. 220–227. IEEE Computer Society / ACM (1996)

[150] Snyder, W., Gallier, J.H.: Higher-order unification revisited: Complete sets of transformations. J. Symb. Comput. 8(1/2), 101–140 (1989)

[151] Steen, A.: Extensional paramodulation for higher-order logic and its effective implementation Leo-III. Ph.D. thesis, Free University of Berlin, Dahlem, Germany (2018)

[152] Steen, A., Benzmüller, C.: There is no best β-normalization strategy for higher-order reasoners. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR-20. LNCS, vol. 9450, pp. 329–339. Springer (2015)

[153] Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. J. Autom. Reason. 65(6), 775–807 (2021)

[154] Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 367–373. Springer (2014)

[155] Subbarayan, S., Pradhan, D.K.: NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 276–291. Springer (2004)

[156] Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and Satallax on the Sledgehammer test bench. J. Appl. Log. 11(1), 91–102 (2013)

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

[158] Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Comm. 26(2), 211–223 (2013)

[159] Sutcliffe, G.: The CADE-26 automated theorem proving system competition—CASC-26. AI Commun. 30(6), 419–432 (2017)

[160] Sutcliffe, G.: The CADE-27 automated theorem proving system competition - CASC-27. AI Commun. 32(5-6), 373–389 (2019)

[161] Sutcliffe, G.: The 10th IJCAR automated theorem proving system competition - CASC-J10. AI Commun. 34(2), 163–177 (2021)

[162] Sutcliffe, G., Suttner, C.B.: Evaluating general purpose automated theorem proving systems. Artif. Intell. 131(1-2), 39–54 (2001)

[163] Suttner, C.B., Sutcliffe, G.: The design of the CADE-13 ATP system competition. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE-13. LNCS, vol. 1104, pp. 146–160. Springer (1996)

[164] Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society s2-42(1), 230–265 (1937)

[165] Turner, D.A.: Another algorithm for bracket abstraction. J. Symb. Log. 44(2), 267–270 (1979)

[166] Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reason. 50(2), 229–241 (2013)

[167] Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer (2014)

[168] Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification 167, 5:1–5:17 (2020)

[169] Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 192–210. Springer (2019)

[170] Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: Fontaine, P., Korovin, K., Kotsireas, I.S., Rümmer, P., Tourret, S. (eds.) PAAR+SC-Square 2020. CEUR Workshop Proceedings, vol. 2752, pp. 148–166. CEUR-WS.org (2020)

[171] Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 316–334. Springer (2020)

[172] Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving (2021), `https://matryoshka-project.github.io/pubs/saturate_article.pdf`, accepted in J. Autom. Reason.

[173] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer (2009)

[174] Wisniewski, M., Steen, A., Kern, K., Benzmüller, C.: Effective normalization techniques for HOL. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 362–370. Springer (2016)

[175] Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM 12(4), 536–541 (1965)

Titles in the IPA Dissertation Series since 2019

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming.* Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09

D.Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-10