Stronger SMT Solvers for Proof Assistants : Proofs, Quantifier Simplification, Strategy Schedules

Hans-Jörg Schurr

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Stronger SMT Solvers for Proof Assistants

Proofs, Quantifier Simplification, Strategy Schedules

THÈSE

présentée et soutenue publiquement le 7 Octobre 2022

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Hans-Jörg Schurr

Composition du jury

Président :	Frédéric BLANQUI
Rapporteurs :	Frédéric BLANQUI Elaine PIMENTEL
Examinateurs :	Chantal KELLER Christophe RINGEISSEN Cesare TINELLI
Encadrants :	Jasmin BLANCHETTE Pascal FONTAINE Stephan MERZ

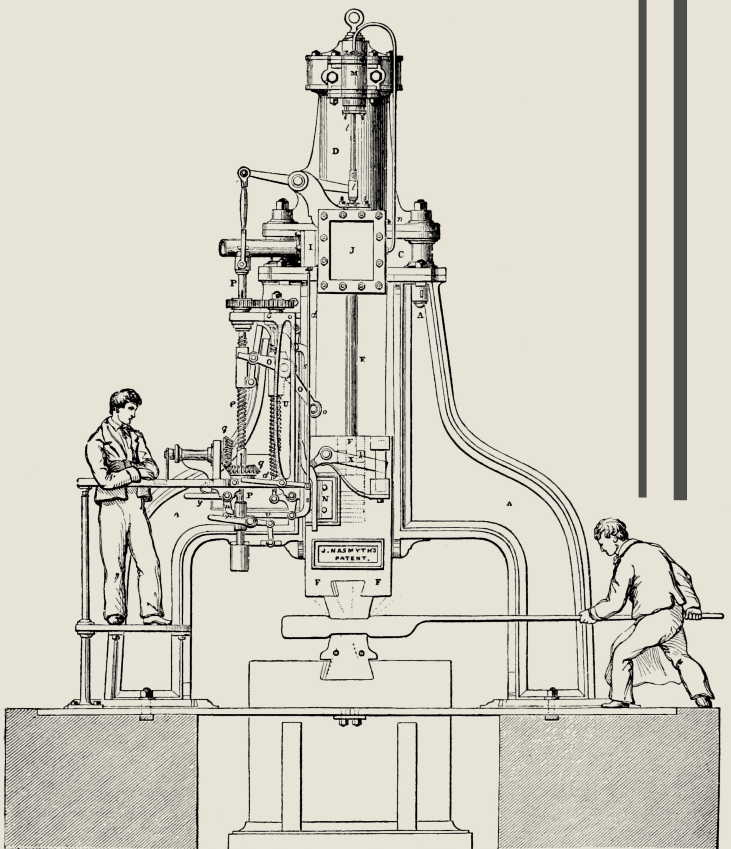
Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Stronger SMT Solvers for Proof Assistants

Proofs, Quantifier Simplification, Strategy Schedules

Hans-Jörg Schurr

PhD Thesis
Université de Lorraine



For my father
Karl-Jürgen Schurr

When you have a moment, practice
so you can fade in a thought from man to tree.
Sometimes I forget to change back.
I watch the mountains fade into twilight
and the stars push through the darkness
like jewels pushing through stone.

—Patricia A. McKillip, The Riddle Master of Hed

Abstract

This thesis presents three contributions that have the same underlying motivation: to improve the utility of SMT solvers as backends for proof assistants. SMT solvers are automated theorem provers that combine propositional reasoning with theories. Proof assistants are tools that empower users to write formally checked proofs. To help the user, proof assistants can provide automation by integrating with automated theorem provers.

Proof assistants typically accept only proofs that are constructed in the trusted kernel of the assistant. The first contribution addresses the reconstruction of SMT proofs in a proof assistant. We present the Alethe proof format for SMT solvers. It improves and unifies previous work on proof generation from SMT solvers. The vast majority of these improvements were informed by the experience provided by a concrete effort: the implementation of a proof reconstructing method in the proof assistant Isabelle/HOL that reconstructs Alethe proofs.

SMT problems generated by proof assistants usually rely heavily on quantifiers. Since SMT solvers excel on quantifier-free problems, they use quantifier instantiation to generate quantifier-free formulas. The second contribution improves quantifier instantiation. It is a unification-based method that augments the problem with shallow quantified formulas obtained from assertions with nested quantifiers. These new formulas help unlocking the regular instantiation techniques, but parsimony is necessary since they might also be misguiding. The method allows the solver to prove more formulas, faster.

The heuristics of an SMT solvers can be parameterized. A specific parameterization of the entire solver is called a strategy, and the best strategy usually differs from problem to problem. The third contribution is a toolbox to work with strategy schedules. A key component of the toolbox is a tool that uses integer programming to generate strategy schedules. Beyond this tool, the toolbox also contains tools for simulating and analyzing schedules. We used the toolbox to select strategies that solve many problems generated by Isabelle/HOL.

Contents

Acknowledgments 5

I Introduction 7
1.1 Publications 9

II Satisfiability Modulo Theories 12
2.1 The Logic of SMT Problems 12
2.1.1 Terms and Sorts 12
2.1.2 The Meaning of Formulas 16
2.1.3 Theories 18
2.1.4 Convenient Notations 23
2.2 Solving SMT Problems 26
2.2.1 The Abstract CDCL(T) Calculus 31
2.2.2 Quantifier Instantiation Techniques 33

III Improving Proofs 39
3.1 The Alethe Proof Format 40
3.1.1 The Alethe Language 41
3.1.2 The Alethe Rules 61
3.2 Reconstructing Alethe Proofs in Isabelle 64
3.2.1 Overview of the veriT-Powered SMT Tactic 65
3.2.2 Tuning the Reconstruction 69
3.2.3 Evaluation 77
3.3 Conclusion and Outlook 88
3.3.1 Related Work 88
3.3.2 Remaining Issues with the Reconstruction 90
3.3.3 Future Developments of Alethe 91

IV Improving Quantifier Simplification 95
4.1 Reasoning With Quantifiers 95
4.1.1 Instantiation Fails 97
4.2 Quantifier Simplification by Unification 99
4.2.1 The Core Rule 100
4.2.2 The Simplification within the SMT Solver 102
4.3 Variants of Quantifier Simplification by Unification 104
4.4 Implementation 105

4.4.1 Indexing and Unification without Skolemization 105
4.5 Evaluation 108
4.5.1 Baseline Comparison 109
4.5.2 Strategy Scheduling 112
4.6 Proof Production 115
4.7 Conclusion 116

V A Toolbox for Strategy Schedules 118
5.1 Integer Programming 119
5.2 Schedule Generation as Integer Programming 120
5.3 Strategy Order and Combined Schedules 123
5.3.1 An Improved Best Effort Order 124
5.3.2 Combining Schedules 125
5.4 The schedgen Toolbox 125
5.4.1 A schedgen Tutorial 128
5.5 Evaluation 132
5.6 Related Work and Outlook 139

VI Conclusion and Outlook 146

Résumé 150

The Alethe Proof Rules 160

Bibliography 193

Index 203

Colophon 206

Acknowledgments

This thesis is a milestone on a long and winding path. I am indebted to many people that helped me to reach this point.

First and foremost my gratitude goes to my advisors. Pascal Fontaine introduced me to the wonderful world of SMT solving. Without his persistent support I would not have found my way through the labyrinth of academic research. Jasmin Blanchette helped me to keep the big picture in mind. His excellent advice allowed me to make many steps forward and to avoid missteps. His Matryoshka project financed most of my time as a PhD student. By running the VeriDis team with wisdom and patience Stephan Merz created the best work environment a PhD student can hope for.

Working in Nancy and in the international formal methods community meant that I was surrounded by many inspiring colleagues. I will remember the time spent with heated discussions after seminars and relaxed conversations over coffee. I am grateful for the wealth of good suggestions they provided me with.

Mathias Fleury implemented the reconstruction of veriT's proofs in Isabelle/HOL. His impressive work informed many improvements to the proof format and justified a significant part of this thesis.

The regular participants of the #nunchaku IRC channel Haniel Barbosa, Guillaume Bury, Simon Cruanes, and Martin Riener were a constant source of inspiration and taught me much about the finer details of SMT solving.

Daniel El Ouraoui helped me a great deal to adapt to live in France and to work at LORIA.

Furthermore, I thank Étienne André, Bruno Andreotti, Alexander Bentkamp, Horatiu Cirstea, Rosalie Defourné, Martin Desharnais, Vincent Despre, Sophie Drouot, Marie Duflot-Kremer, Margaux Duroeulx, Yann Duplou, Alexis Grall, Igor Konnov, Pierre Lermusiaux, Matthieu Nicolas, Mathias Preiner, حمید ره کوی, Christophe Ringeissen, Sorin Stratulat, Sophie Tourret, Petar Vukmirović.

During the long years of my studies, I could always count on the support of my friends. With some I had the pleasure to share drinks, with some I spent hours discussing life, and yet others offered their couch while I was traveling. Thank you Guilherme Alves, Aaron Berthold, Bir, , Jakob Bleier, بۆکان, Christine Boucher, Charlie, , Lisa Chedik, Alvin Chiang, Lucia Dangel, Mathias Ertl, Mathias Fassel, Laétitia Franz, metamatik, Julian Golderer, Noémie

Gonnier, Hannah, Jean-Marc Henry, Jo, Jonas, Kevin, سيليا, George Krait, Анна Кравченко, Valerie Lauer, 이승훈, Ксенія Максімава, Kaja Merle, Dominique Mias-Lucquin, Noam, Lucie Pieters, Gregor Plieschnig, Véronique Sotton, Richard Sung, the Telekonvicts, Carl Vaginay, Kristina Weinberger, Γιώργος Ζερβάκης.

Athénaïs Vaginay is a constant source of joy and support in my life. I will never forget this.

Finally, pursuing this thesis far from home would not have been possible without the understanding and encouragement from my family. I send greetings to my brother Christoph Schurr. There are no words to express my gratitude to my mother, Christine Schurr.

Jasmin Blanchette, Pascal Fontaine, Pierre Lermusiaux, Martin Riener, Sophie Tourret, and Athénaïs Vaginay suggested improvements to the text. Thank you!

I am grateful to the people who agreed to dedicate time to the evaluation of my thesis: Frédéric Blanqui as director of the jury and rapporteur; Elaine Pimentel as rapporteur; Chantal Keller, Christophe Ringeissen, and Cesare Tinelli as members of the jury. Christophe Ringeissen also provided continuous feedback on the progress of my thesis through his role as référent.

> You said: “the ship in port is the safer one,
> but it's not the reason it was made.”
> So forgive me if I wander off,
> and forgive me more if I just stay.
>
> —Radical Face, The Ship in Port

I Introduction

This thesis presents three contributions that have the same underlying motivation: to improve the utility of SMT solvers as backends for proof assistants. SMT solvers are a type of fully automated theorem provers. They combine propositional reasoning with theories and excel at problems that do not use quantifiers in an extensive manner (see Chapter II). In this thesis, we use the SMT solver veriT [31]. Proof assistants, also called interactive theorem provers, are tools that empower users to write formally checked proofs. They provide sophisticated mechanisms to work with deep mathematical theories or complicated verification problems.

Proof assistants should make the life of the user easy. The user should be able to focus on the tricky parts of their proof and not get distracted by the technical aspects of the assistant. One way to ensure this is to provide automation. Automation can be implemented wholly within the proof assistant, or use external tools, such as SMT solvers. When automation works well, proof obligations that are "trivial" can be taken care of automatically. A

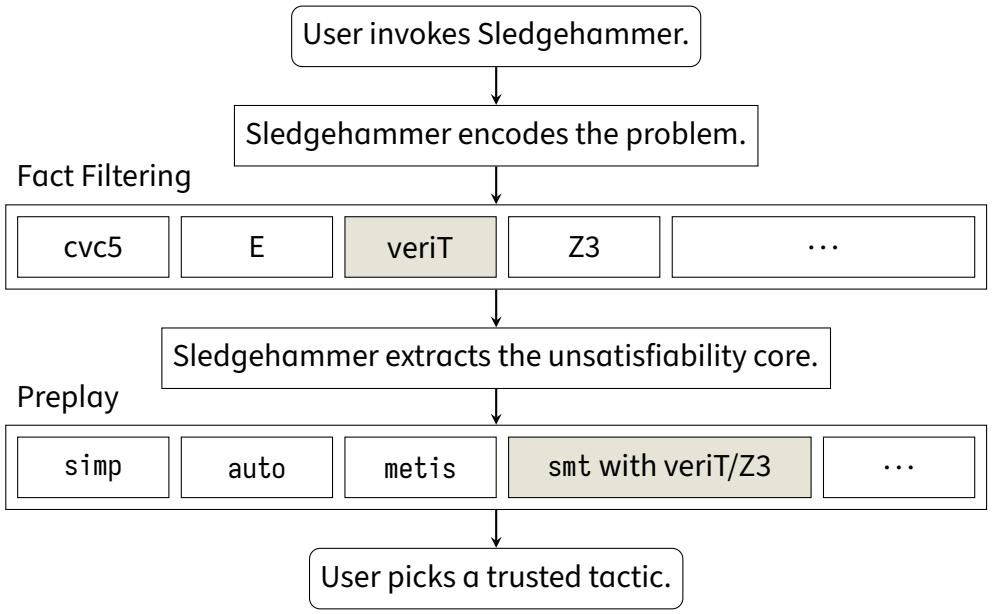


Figure I.1 The interaction between the user, Sledgehammer, and automated theorem provers.

concrete example of such an integration is the Sledgehammer [24] tool that is part of the Isabelle/HOL [22] proof assistant.

Figure I.1 shows the Sledgehammer pipeline. Sledgehammer is invoked by the user using the Isabelle/HOL interface. On invocation, Sledgehammer performs two steps: fact filtering and preplay. To start fact filtering, Sledgehammer encodes the current proof obligation and selected facts from the background theories into the logic of the automated theorem provers. Then, it invokes external automated theorem provers according to a schedule. This step succeeds if at least one of the automated theorem provers finds a proof. However, Sledgehammer cannot use this proof directly. Isabelle/HOL and many other proof assistants only accept proofs that are expressed in terms of their trusted reasoning kernel.

To find such a proof, Sledgehammer performs the second step: *preplay*. It again invokes automated tools according to a schedule. However, this time it invokes internal tools that use the trusted kernel, such as the simplifier `simp`. To help the internal tools to solve the proof obligation, Sledgehammer uses the unsatisfiability core extracted from the first step. It only provides the facts in the core to the internal tools. Therefore, the automated theorem provers act as fact filters in the first step. If an internal tool is successful during preplay, it is proposed to the user, and the user can incorporate it into their proof. The `smt` tactic is a special case of an internal tool: it also calls an external automated theorem prover, but retrieves a full proof from them and reconstructs the proof within the trusted kernel. If Sledgehammer often succeeds, the user is often happy.

A failure case that is particularly frustrating is a failure of the preplay step: the user knows that there is a proof, but they cannot use it. One way to reduce the rate of this type of failure is to expand the `smt` tactic to replay different proofs. How easy it is to implement a procedure that performs this replay and how reliable it is depends to a high degree on the quality of the proof generated by the automated theorem prover. Chapter III addresses this subject from the side of the SMT solver. First, Section 3.1 discusses the Alethe proof format. This format improves and unifies previous work on proof generation from SMT solvers. The vast majority of these improvements were informed by the experience provided by a concrete effort of expanding the `smt` tactic with support for Alethe proofs. This effort is documented in Section 3.2.

To allow the user to express complex mathematical theorems, proof assistants use expressive logics. In the case of Isabelle/HOL, this is higher-order logic. In contrast, SMT solvers focus on reasoning on quantifier-free first-order problems. To bridge the gap, the main approach right now is for Sledgehammer to encode the proof obligation into first-order logic and for the SMT solver to generate quantifier-free instances of first-order formulas. The contribution discussed in Chapter IV improves this pipeline. It adds a preprocessing step to the SMT solver that can eliminate nested quantifiers by using unification.

In practice, automated theorem provers must use heuristics to solve many problems. SMT solvers do not differ from other systems in this regard. Typically, heuristics can be parameterized. A specific parameterization of an SMT solver is called a strategy, and the best strategy usually differs from problem to problem. Hence, an SMT solver user can profit from using an appropriate strategy and from trying multiple strategies. The third contribution, presented in Chapter V, is a toolbox to work with strategy schedules. A key component of the toolbox is a tool that uses integer programming to generate strategy schedules. Beyond this tool, the toolbox also contains tools for simulating and analyzing schedules. We used this tool to find strategies that work well with Sledgehammer.

Overall, this thesis shows that when an SMT solver is treated as a complex software system instead of a black-box implementation of a fixed calculus, many avenues to improve it open up (Chapter VI).

1.1 Publications

Some material included in this thesis was published before. The inclusion here has the consent of all co-authors. I adapted the notation and some text to ensure consistency throughout the thesis and added some material that was omitted from the published version due to space constraints.

Contribution 1 Section 3.1 is based on a document that was previously published informally. Originally, called “Proofonomicon” and was entirely written by me. Later, the name changed to “The Alethe Proof Format: A Speculative Specification and Reference” and Haniel Barbosa, Mathias Fleury, and Pascal Fontaine contributed to it. Furthermore, Bruno Andreotti, Hanna Lachnitt, Chantal Keller, and Arjun Viswanathan provided feedback.

The description of the future directions for the Alethe proof format in Section 3.3.3 contains some paragraphs of the following publication.

H.J. Schurr, M. Fleury, H. Barbosa, and P. Fontaine, Alethe: Towards a Generic SMT Proof Format (extended abstract), In PxTP 2021 – 7th Workshop on Proof eXchange for Theorem Proving (Pittsburgh, PA / virtual, United States, 2021).

Furthermore, the following publication relates to proof generation from veriT, but was not incorporated into the text.

H. Barbosa, J.C. Blanchette, M. Fleury, P. Fontaine, and H.J. Schurr, Better SMT Proofs for Easier Reconstruction, In AITP 2019 – 4th Conference on Artificial Intelligence and Theorem Proving (Obergurgl, Austria, 2019).

Section 3.2 and parts of Section 3.3.2 are based on the publication

H.J. Schurr, M. Fleury, and M. Desharnais, Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant, In A. Platzer and G. Sutcliffe (Eds.) Automated Deduction – CADE 28 (Springer International Publishing, Cham, 2021).

The work described in this publication was performed in tight collaboration with Mathias Fleury. He implemented the reconstruction in Isabelle/HOL and I improved the proof format and proof output generated by veriT to ensure it is easy to reconstruct. Martin Desharnais contributed to the experiments and performed much of the analysis of the experimental data that we obtained. Haniel Barbosa, Jasmin Blanchette, Pascal Fontaine, Daniela Kaufmann, Petar Vukmirović, and anonymous reviewers provided feedback.

Furthermore, Figure 3.4 and description of the figure, Section 3.2.1.1, and Section 3.2.1.2 were taken from

M. Fleury and H.J. Schurr, Reconstructing veriT Proofs in Isabelle/HOL, In G. Reis and H. Barbosa (Eds.) Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, Natal, Brazil, August 26, 2019 (Open Publishing Association, 2019).

Alex Figl-Brick, Daniel El Ouraoui, Pascal Fontaine, and anonymous reviewers provided feedback.

Contribution 2 Chapter IV is based on the publication

P. Fontaine and H.J. Schurr, Quantifier Simplification by Unification in SMT, In B. Konev and G. Reger (Eds.) Frontiers of Combining Systems - 13th International Symposium (Springer International Publishing, Cham, 2021).

Haniel Barbosa, Jasmin Blanchette, Antoine Defourné, Daniel El Ouraoui, Mathias Fleury, Martin Riener, Athénaïs Vaginay, and anonymous reviewers provided feedback.

Contribution 3 Chapter V was, with the exception of Section 5.6, published as

H.J. Schurr, Optimal Strategy Schedules for Everyone, In B. Konev, C. Schon, and A. Steen (Eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning (CEUR-WS.org, 2022).

Pascal Fontaine and the anonymous reviewers provided feedback.

II Satisfiability Modulo Theories

This chapter serves two purposes. On the one hand, it introduces the background common to all subsequent chapters. On the other hand, it introduces the notations used throughout the thesis.

Section 2.1 provides the bedrock for the rest of the thesis. It introduces the SMT-LIB logic. This logic is used by most SMT solvers, and in particular by the SMT solver veriT – the SMT solver we concern ourselves with. In Subsection 2.1.3 we discuss some common SMT theories. This section finishes with Subsection 2.1.4 that introduces some useful notations and concepts. The second section is more concrete. Section 2.2 introduces the CDCL(T) calculus. Based on the successful CDCL calculus used in SAT-solving, the CDCL(T) calculus forms the basis for most SMT solvers. Finally, Subsection 2.2.2 discusses quantifier instantiation, which is used by SMT solvers to reason with quantifiers.

2.1 The Logic of SMT Problems

A major achievement of the SMT community is that many solvers support the same logic and input language. We now introduce this logic, which is also the logic used throughout this thesis. It is part of the SMT-LIB library. This library defines not only a logic and a language, but also defines common theories and provides an extensive benchmark collection. We use precisely the logic and concrete language of the SMT-LIB library, but will take some liberties with respect to the abstract language when appropriate.

The SMT-LIB logic is a many-sorted first-order logic with equality. Part 3, Chapter 5 of the SMT-LIB standard document [17] provides a precise abstract syntax and semantic of the SMT-LIB many-sorted first-order logic. For a general overview of many-sorted logic, see, for example, a book chapter by Manzano [69]. We will limit ourselves to a simplified abstract syntax. Many SMT-LIB features, such as parametric sorts, and matching, are not relevant for this work and are omitted.

2.1.1 Terms and Sorts

The term language of the SMT-LIB logic uses mutually disjoint sets of predefined symbols. Each such set is associated with a character that serves as a

generic placeholder for the symbols in the set. It can also be sub-scripted. For example, f, f_1, f_n all stand for function symbols. Predefined symbols are printed in bold sans-serif font (e.g., **Bool** for the sort of Booleans) in the abstract syntax.

- The set $\mathcal{S}$ contains the sort symbols denoted by σ. The set contains the symbol **Bool** for Booleans.
- The infinite set $\mathcal{X}$ contains the variables denoted by x and y.
- The set $\mathcal{F}$ contains the function symbols denoted by f, f_1, … This set contains the symbols $\approx$, $\wedge$, $\vee$, and $\neg$.
- The two-element set $\mathcal{B} = \{\mathsf{true}, \mathsf{false}\}$ contains the symbols for the Boolean values. We write b for either Boolean value.

SMT-LIB *terms* are constructed from the function symbols and variables taken from their respective sets together with the three binders $\forall$, $\exists$, and **let**…**in**….

Definition 1 (Terms)
The following grammar rule describes the construction of terms t.

$$t ::= t \mid x \mid f\ t^* \mid f^\sigma\ t^* \mid \exists\,(x:\sigma)^+.\,t \mid \forall\,(x:\sigma)^+.\,t \mid \mathsf{let}\,(x=t)^+\,\mathsf{in}\,t$$

The notation $(\ )^*$ denotes a possibly empty and finite list, and $(\ )^+$ a finite list with at least one member.

This term language differs from the familiar terms of first-order logic. First, function application is written in an λ-application style. Instead of $f(x, g(y))$ we write $(f\ x\ (g\ y))$. Second, function symbols can be overloaded. Since the SMT-LIB logic is many-sorted, every term has an associated sort. To avoid ambiguous sorts, there is a syntax to concretize the sort of a function symbol: f^σ. Third, this term language also encompasses terms with Boolean sort – there is no distinction between terms and formulas. Finally, symbols such as equality and the Boolean connectives are not part of the syntax. They are handled by the special theory *Core*.

Example 1
Some terms: $(+\ (\times\ 3\ 4)\ 5)$, $(+^{\mathsf{Int}}\ (\times\ 3\ 4)\ 5)$, $\exists\,x : \mathsf{Int}.\,(\geq\ x\ 5)$, $\exists\,x : \mathsf{Int}\ y : \mathsf{Int}.\,(\geq\ (+\ x\ y)\ 5)$, $\forall\,x : \mathsf{Int}.\,(\exists\,y : \mathsf{Int}.\,(>\ y\ x))$,

In general, we only work with *well-sorted* terms. Sorts are associated to terms by sorting rules. Semantically, each sort represents a set of values from the universe. A well-sorted term is a term for which a sorting derivation exists.

The sorting rules use a signature. The signature determines the concrete function symbols and sorts available to build well-sorted terms. The concrete signature depends on the chosen theories and can also be extended by the user with uninterpreted sorts and symbols by invoking commands to do so. SMT-LIB supports parametric sorts and sort terms, but we do not use the features.

Definition 2 (Signature)
A *signature* is a tuple $\Sigma = (\Sigma^S, \Sigma^F, V, R)$ consisting of

- a finite set $\Sigma^S \subseteq \mathcal{S}$ of sort symbols that contains the symbol **Bool**,
- a set $\Sigma^F \subseteq \mathcal{F}$ of function symbols,
- a partial mapping V from $\mathcal{X}$ to Σ^S that assigns sorts to some variables,
- and a *ranking* relation $R \subseteq \Sigma^F \times (\Sigma^S)^+$. The relation must be *left-total*: for all $f \in \Sigma^F$ there is a $\eta \in (\Sigma^S)^+$ such that $f\,R\,\eta$. The set $(\Sigma^S)^+$ is the set of non-empty finite lists over the sort symbols in Σ^S.

The lists $(\Sigma^S)^+$ are the *ranks*. From left to right, a rank denotes the expected sort of the symbol's arguments and of the result. Since the SMT-LIB logic allows *overloading*, R is not left-unique. The form of overloading supported by the SMT-LIB logic is called *ad hoc polymorphism*.

Ad hoc polymorphism makes it possible to use a function symbol with different ranks. For example, for the equality symbol "$\approx$", that is part of the Core theory, $\approx\, R\, \sigma\sigma$**Bool** holds for all sort symbols $\sigma \in \Sigma^S$. The definition does not even restrict the arity of a function symbol. In a theory of arithmetic we can have a function symbol "$-$" with both the ranks **Int Int** and **Int Int Int**. In this case, the first rank is used when "$-$" serves as unary negation, and the second one is used for the usual subtraction. A function symbol together with a sort is called a *ranked function symbol*. Formally, a ranked function symbol is a tuple $(f, \sigma_1 \ldots \sigma_n\sigma) \in \Sigma^F \times (\Sigma^S)^+$, which we write as $f : \sigma_1 \ldots \sigma_n\sigma$. In the same spirit, a *sorted variable* is a tuple $(x, \sigma) \in \mathcal{X} \times \Sigma^S$, which we write as $x : \sigma$. We write $f : \sigma_1 \ldots \sigma_n\sigma \in \Sigma$ if $f\,R\,\sigma_1 \ldots \sigma_n\sigma$ and $x : \sigma \in \Sigma$ if $V(x) = \sigma$. Note that the definition of the ranking relation allows some ambiguity. This problem comes from function symbols that are assigned two ranks of the

form $\sigma'\sigma_1$ and $\sigma'\sigma_2$ where σ' is a list of sorts. In this case, a term $f\ t^+$ is ambiguous if the terms t^+ have the sorts σ'. In this case, one has to use the syntax that clarifies sorts. For example, one can write $f^{\sigma_1} t^+$. Since V assigns at most one sort to variables, every term can have at most one sort if the signature contains no such function symbol.

The sorting rules and the semantics are significantly simplified by the fact that the current version 2.6 of SMT-LIB does not allow partial application. For example, (-3) is always the negation of 3, and never a partially applied subtraction. Partial application, however, is standard in higher-order logics. Hence, the upcoming version 3 of SMT-LIB will support partial application, and a solution for these problems must be found.[1]

Overall, the set of well-sorted terms depends on the current signature Σ.

Definition 3 (Well-sorted Term)
A term t generated by the grammar rules in Definition 1 is *well-sorted* for a signature Σ if t *has sort* σ. Let $\Sigma[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$ denote the signature for which the function V maps x_i to σ_i for $i \in 1, \ldots, n$ and otherwise corresponds to V of Σ. The term t has sort σ if a judgment $\Sigma \vdash t : \sigma$ is derivable by the following sorting rules.

$$\frac{}{\Sigma \vdash x : \sigma} \text{ if } x : \sigma \in \Sigma$$

$$\frac{\Sigma \vdash t : \sigma}{\Sigma \vdash (t) : \sigma}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f\ t_1 \cdots t_k) : \sigma} \text{ if } \begin{cases} f : \sigma_1 \cdots \sigma_k \sigma \in \Sigma, \\ f : \sigma_1 \cdots \sigma_k \sigma' \notin \Sigma \text{ for all } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_k : \sigma_k}{\Sigma \vdash (f^\sigma\ t_1 \cdots t_k) : \sigma} \text{ if } f : \sigma_1 \cdots \sigma_k \sigma \in \Sigma$$

$$\frac{\Sigma[x_1 : \sigma_1, \ldots, x_{k+1} : \sigma_{k+1}] \vdash t : \textbf{Bool}}{\Sigma \vdash (Q\ x_1 : \sigma_1 \cdots x_{k+1} : \sigma_{k+1}.\ t) : \textbf{Bool}} \text{ if } Q \in \{\forall, \exists\}$$

[1] The current proposal is available at https://smtlib.cs.uiowa.edu/version3.shtml.

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \cdots \quad \Sigma \vdash t_{k+1} : \sigma_{k+1} \qquad \Sigma[x_1 : \sigma_1, \ldots, x_{k+1} : \sigma_{k+1}] \vdash t : \sigma}{\Sigma \vdash (\textsf{let}\ x_1 = t_1 \cdots x_{k+1} = t_{k+1}\ \textsf{in}\ t) : \sigma} \text{ if } x_1, \ldots, x_{k_1} \text{ are all distinct}$$

Definition 4 (Formulas and Sentences)
For a given signature Σ, the language of SMT-LIB logic is the set of all well-sorted terms with respect to Σ. *Formulas* are the well-sorted terms of sort **Bool**.
A formula without free variables is called a *sentence*.[2] A term without any variables is called a *ground term*. A formula that is a ground term is a *ground formula*.

Example 2
The term $(\approx (+\ 5\ 3)\ x)$ is a formula, but not a sentence, since x is free. The term $\exists\, x : \textsf{Int}.\ (\approx (+\ 5\ 3)\ x)$, on the other hand, is a sentence.

2.1.2 The Meaning of Formulas

To define the semantic of the SMT-LIB logic, we use the familiar notion of Σ-structure.[3]

Definition 5 (Σ-structure)
Let Σ be a signature. A Σ-structure is a tuple $\mathcal{A} = (U, I)$ of a set U, the universe, and a mapping I.

- Each sort $\sigma \in \Sigma^S$ is mapped to a non-empty set $I(\sigma) \subseteq U$, the *domain* of σ. The mapping is such that $U = \bigcup_{\sigma \in \Sigma^S} I(\sigma)$.
- For each sort $\sigma_1 \ldots \sigma_n \sigma$, $I(\sigma_1 \ldots \sigma_n \sigma)$ is the set of total-functions from $I(\sigma_1) \times \ldots \times I(\sigma_n)$ to $I(\sigma)$.
- $I(\textsf{Bool}) = \{\textsf{true}, \textsf{false}\}$, where **true** and **false** are distinct and both are elements of U.

[2] See Section 2.1.4 for the formal definition of free variables.
[3] The SMT-LIB specification distinguishes between ordinary Σ-structures and SMT-LIB Σ-structures. Since we ignore datatypes those are almost equivalent for us and we ignore the distinction.

- Each ranked function symbol $f : \sigma$ (i.e., each constant) is mapped such that $I(f : \sigma) \in I(\sigma)$.
- Each ranked function symbol $f : \sigma_1 \ldots \sigma_n \sigma \in \Sigma$ is mapped to a total function from $I(\sigma_1 \ldots \sigma_n \sigma)$.

This definition does not require the domains of the sorts to be disjoint. This is due to the sorting of equality. No term $(\approx t_1\ t_2)$ is well sorted if the terms t_1 and t_2 have different sorts. Hence, if a formula is satisfiable in a Σ-structure, it is also satisfiable in a structure where all domains are disjoint.

A Σ-structure does not handle assignments to variables. This is handled by a *valuation into a* Σ*-structure*. It is a mapping v that assigns values to ranked variables. More precisely, it is a partial mapping from $\mathcal{X} \times \Sigma^S$ to elements from U such that $v(x : \sigma) \in I(\sigma)$. Furthermore, for a valuation v, the valuation $v[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n]$ is the function that corresponds to the valuation v except for the values assigned to the variables $x_1 : \sigma_1, \ldots, x_n : \sigma_n$; those variables are mapped to the corresponding a_i.

A Σ-*interpretation* $\mathcal{I}$ is a pair $(\mathcal{A}, v)$ of a Σ-structure $\mathcal{A}$ and a valuation v. We write $\mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n]$ as a shorthand for the Σ'-interpretation

$$(\mathcal{A}', v[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n])$$

where Σ' is $\Sigma[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$ and $\mathcal{A}'$ is $\mathcal{A}$ seen as a Σ'-structure.

We can naturally extend a Σ-interpretation into a unique and total mapping from well-sorted Σ-terms to values from the universe of Σ. This mapping is denoted $[\![.]\!]^{\mathcal{I}}$. It is what assigns "meaning" to terms.

Definition 6
Let Σ be a signature and $\mathcal{I}$ a Σ-interpretation. For every well-sorted term t with the sort σ the mapping $[\![t]\!]^{\mathcal{I}}$ is defined recursively as follows.

1. $[\![x]\!]^{\mathcal{I}} = v(x : \sigma)$ where $\mathcal{I} = (\mathcal{A}, v)$

2. $[\![\hat{f}\ t_1 \ldots t_n]\!]^{\mathcal{I}} = \mathcal{A}(f : \sigma_1 \ldots \sigma_n \sigma)(a_1, \ldots, a_n)$
   where $\begin{cases} \mathcal{I} = (\mathcal{A}, v) \text{ with signature } \Sigma, \\ \hat{f} = f \text{ or } \hat{f} = f^{\sigma} \\ \text{for } i = 1, \ldots, n \\ \quad \Sigma \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}} \end{cases}$

3. $[\![\textsf{let } x_1 = t_1 \cdots x_n = t_n \textsf{ in } t]\!]^{\mathcal{I}} = [\![t]\!]^{\mathcal{I}'}$
$$\text{where} \begin{cases} \mathcal{I} \text{ has signature } \Sigma, \\ \text{for } i = 1, \ldots, n \\ \quad \Sigma \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}}, \\ \mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n] \end{cases}$$

4. $[\![\exists x_1 : \sigma_1 \cdots x_n : \sigma_n . t]\!]^{\mathcal{I}} = \textsf{true}$ iff $[\![t]\!]^{\mathcal{I}'} = \textsf{true}$ for some $\mathcal{I}'$
$$\text{where} \begin{cases} \mathcal{I} = ((U, I), v), \\ (a_1, \ldots, a_n) \in I(\sigma_1) \times \cdots \times I(\sigma_n), \\ \mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n] \end{cases}$$

5. $[\![\forall x_1 : \sigma_1 \cdots x_n : \sigma_n . t]\!]^{\mathcal{I}} = \textsf{true}$ iff $[\![t]\!]^{\mathcal{I}'} = \textsf{true}$ for all $\mathcal{I}'$
$$\text{where} \begin{cases} \mathcal{I} = ((U, I), v), \\ (a_1, \ldots, a_n) \in I(\sigma_1) \times \cdots \times I(\sigma_n), \\ \mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n] \end{cases}$$

The notion of a satisfiable or falsifiable formula is defined as usual. A Σ-interpretation *satisfies* a formula φ if $[\![\varphi]\!]^{\mathcal{I}} = \textsf{true}$. If, on the other hand, $[\![\varphi]\!]^{\mathcal{I}} = \textsf{false}$, then the Σ-interpretation *falsifies* the formula. Furthermore, a formula φ is *satisfiable* if there is a Σ-interpretation that satisfies it and, accordingly, it is *unsatisfiable* if there is none.

The notion of a *model* only applies to sentences. In this case, the valuation v is irrelevant. Hence, a model of a formula without free variables φ is a Σ-structure $\mathcal{A}$ such that every (or, equivalently, at least one) Σ-interpretation $(\mathcal{A}, v)$ satisfies φ. As usual, $\models$ denotes semantic entailment. A set of sentence Γ entails a sentence ψ iff every model that satisfies all sentences in Γ satisfies ψ as well. This is denoted $\Gamma \models \psi$ and if Γ contains only one sentence φ it is denoted $\varphi \models \psi$.

2.1.3 Theories

Obviously, reasoning with theories is an integral part of SMT solving. For us, however, it is not of central interest. In this thesis, we will only encounter some simple standard theories. Those are the theory of Booleans and uninterpreted functions (the Core theory), and linear arithmetic with reals and integers. Let us first have a look at how the SMT-LIB handles theories and then discuss those theories.

Definition 7 (Theory)
A Σ-*theory* $\mathcal{T}$ is a class of Σ-structures on a signature Σ. These structures are the *models* of the theory $\mathcal{T}$.

Defining theories this way gives us the flexibility to declare theories as either specific models (such as the usual model of natural numbers), or as the Σ-structures that satisfies a set of sentences which are called axioms.

We are now finally able to say what it means for a formula to be satisfiable *modulo* – or simply *in* – a theory.

Definition 8 (Satisfiability and Entailment In a Theory)
A Σ-sentence is satisfiable in a theory $\mathcal{T}$ iff it is satisfied by one of the models of $\mathcal{T}$. A set Γ of Σ-sentences $\mathcal{T}$-entails a Σ-sentence φ (denoted $\Gamma \vDash_{\mathcal{T}} \varphi$) iff every model of $\mathcal{T}$ that satisfies all sentences in Γ satisfies φ as well.

The SMT-LIB language allows the combination of theories. Theories can be combined if their signatures are compatible. Signatures are compatible if they have the same sort symbols and agree on the sorts they assign to variables. Since the SMT-LIB language supports polymorphic function symbols, the signatures do not have to agree on the sorts assigned to function symbols. The result of combining two compatible theories is as expected: it is the class of Σ-structures whose projections to the signature of a component theory are models of that theory.

We typically want to combine theories that do not share all sort symbols (e.g., the theory of Booleans and integer arithmetic). In this case we can add the missing sorts to the the signatures of the involved theories. The models of these extended theories are Σ-structures whose projections to the original signature are models of the original theory. The domains of the added sorts are unrestricted. It is possible to combine multiple theory solvers by using a theory combination procedure, such as the Nelson-Oppen method [75]. The combination of theory solvers has many subtleties and is a field of research all in itself. Here, we will not discuss theory combination methods.

Within SMT-LIB parlance, a *logic* is a specific signature Σ together with a theory $\mathcal{T}$ and a restriction to the set of allowed sentences. The restrictions are usually syntactical and ban undefined sentences. For example, in the logic of linear arithmetic, it is pointless to multiply two variables. We will follow the common practice of the SMT community and do not always draw a clear distinction between “logic” and “theory”.

It is possible to use any logic with and without quantifiers. If a logic is specified to be *quantifier-free*, sentences cannot contain terms of the form $Q\, x_1 : \sigma_1 \cdots x_n : \sigma_n . t$ where $Q \in \{\forall, \exists\}$. Since SMT solving is historically concerned with answering the satisfiability of variable-free formulas, quantifier-free logics are widely used. Since we are concerned with quantifier instantiation and proofs for interactive theorem proving, we will not restrict ourselves to quantifier-free logics.

The Core theory[4] This theory is used by every SMT-LIB logic. It contains the usual Boolean connective $\wedge$, $\vee$, $\neg$, $\rightarrow$, **xor**. Since SMT-LIB is restricted to ASCII characters, it uses the strings `and`, `or`, `not`, `=>`, and `xor` for those connectives. Equality, denoted by $\approx$ (`=` in SMT-LIB), is also part of this theory. The Core theory does not contain a dedicated "if and only if" connective. Instead equality with Boolean sorts is used. Beside those basic connectives, whose sorts and interpretation should be obvious, there are two further operators. First, **distinct** : $\sigma\sigma$**Bool** is a convenience operator used to denote that two terms are not equal. The structures of the Core theory are the structures that interpret this operator as

$$[\![\textbf{distinct}\, t_1\, t_2]\!]^{\mathcal{J}} = \textbf{false} \text{ if } [\![t_1 \approx t_2]\!]^{\mathcal{J}} = \textbf{true},$$
$$[\![\textbf{distinct}\, t_1\, t_2]\!]^{\mathcal{J}} = \textbf{true} \text{ otherwise.}$$

Using syntactic sugar, SMT-LIB allows **distinct** to be chained and used as an n-ary operator. **distinct** $t_1 \ldots t_n$ is recursively interpreted as

$$(\wedge\, (\textbf{distinct}\, t_1\, t_2) \cdots (\textbf{distinct}\, t_1\, t_n)\, (\textbf{distinct}\, t_2 \ldots t_n)).$$

The second convenience operator is **ite** : **Bool**$\sigma\sigma\sigma$. It is used to represent if-then-else. It is interpreted as

$$[\![\textbf{ite}\, \varphi\, t_1\, t_2]\!]^{\mathcal{J}} = [\![t_1]\!]^{\mathcal{J}} \text{ if } [\![\varphi]\!]^{\mathcal{J}} = \textbf{true},$$
$$[\![\textbf{ite}\, \varphi\, t_1\, t_2]\!]^{\mathcal{J}} = [\![t_2]\!]^{\mathcal{J}} \text{ otherwise.}$$

[4] The official specification of the Core theory is available at https://smtlib.cs.uiowa.edu/theories-Core.shtml .

The theory of linear arithmetic Linear arithmetic within SMT-LIB is defined by the theories for reals and integers[5] together with syntactic restrictions that restrict arithmetic expressions to linear expressions[6].

The signature of the theory of reals and integers contains two additional sorts: **Int** and **Real**. It also contains the function symbols listed in Table 2.1. Most functions are overloaded to work with integers and reals. The interpretation of the theory is as expected. Division, however, is a special case since SMT-LIB does not support partial functions. Hence, terms such as $(/\ x\ 0.0)$ are well-defined. The standard, however, does not constrain the interpretation of such terms. As a consequence, for each real constant v, the term $(\approx v\ (/\ t\ 0.0))$ has a model. In practice, this means that such terms are useless. The same applies for **div** – the integer division.
A subtlety within the SMT-LIB language is the sort of syntactic *numerals*. This syntactic category refers to numbers without a decimal point. For example, `42` is a numeral, but `42.0` is not. Constants such as `42.0` are in the syntactic category *decimal*. Within theories that only use reals, numerals have the sort **Real**. Within theories that only use integers, or both integers and reals, numerals have the sort **Int**. Decimals are of sort **Real** in theories that use reals, and are not sorted in theories that only use integers. Here, we will always use numerals for numbers of sort **Int** and decimals for **Real**.

Linear arithmetic is a syntactic restriction on top of the theory of integers and reals (or integers only, reals only). Terms with sort **Real** or **Int** must not have an occurrence of the function symbols $\times$, $/$, **div**, **mod**, **abs**, except if the term is a monomial.

Definition 9 (Monomial)
A *monomial* is a term of the form $(\times\ c\ x)$ or $(\times\ x\ c)$ where x is a free constant and c is either an integer or a rational coefficient.
An integer coefficient is a term of the form n or $(-\ n)$ where n is a numeral.
A rational coefficient is a term of the form d, $(-\ d)$, or $(/\ c\ n)$ where d is a decimal, c an integer coefficient, n is a numeral other than 0.

[5] Available at https://smtlib.cs.uiowa.edu/theories-Reals_Ints.shtml.
[6] Such syntactic restriction are part of a specific SMT-LIB logic. The only logic specified on the SMT-LIB website that uses linear arithmetic with both reals and integers combines it with quantifiers and arrays (called AUFLIRA). It is available at https://smtlib.cs.uiowa.edu/logics-all.shtml#AUFLIRA. Nevertheless, the restrictions apply similarly to other logics with linear arithmetic.

Symbol	String	Rank	
−	`-`	**Int Int**	negation
		Real Real	
−	`-`	**Int Int Int**	subtraction
		Real Real Real	
+	`+`	**Int Int Int**	
		Real Real Real	
×	`*`	**Int Int Int**	
		Real Real Real	
div	`div`	**Int Int Int**	
/	`/`	**Real Real Real**	
mod	`mod`	**Int Int Int**	
abs	`abs`	**Int Int**	
≤	`<=`	**Int Int Bool**	
		Real Real Bool	
<	`<`	**Int Int Bool**	
		Real Real Bool	
≥	`>=`	**Int Int Bool**	
		Real Real Bool	
>	`>`	**Int Int Bool**	
		Real Real Bool	
to_real	`to_real`	**Int Real**	
to_int	`to_int`	**Real Int**	
is_int	`is_int`	**Real Bool**	
divisible n	`(_ divisible` *n* `)`	**Int Bool**	for positive integers *n*

Table 2.1 Function symbols of the real and integers theory.

Example 3
The terms $(+\ (\times\ 5\ c)\ (\times\ c\ 11))$, $(f\ 5\ (g\ (+\ x\ y)))$, and $(\times\ ((/\ (-\ 5)\ 4))\ c)$ are all admissible terms in linear arithmetic. The terms $(\times\ a\ b)$, $(\times\ (+\ 4\ 3)\ c)$, and $(\times\ (\textbf{div}\ 4\ 2)\ c)$ are not.

In this thesis, linear arithmetic appears in two sections. Section 3.1 discusses the Alethe proof format that can contain proof steps for linear arithmetic and Section 5.1 uses the theory of linear arithmetic as a convenient way to introduce integer programming.

2.1.4 Convenient Notations

Working with terms only in prefix notation is a somewhat tedious task. Furthermore, in the last section some common terminology – such as "free variable" – was used without a proper introduction. This section introduces various concepts and notations that are useful throughout this thesis.

Let us tackle the term notation first. Function symbols that are usual used in infix notation are also used in infix notation by us. For example, $(+\ a\ b)$ becomes $(a + b)$. This applies to the symbols $\wedge$, $\vee$, $\rightarrow$, **xor**, $+$, $\times$, $-$, **div**, $/$, **mod**, $\leq$, $<$, $\geq$, $>$. Furthermore, we drop superfluous brackets around associative operators. The term $1 + (2 + 3)$ becomes $1 + 2 + 3$. We also write negation like constants, i.e., $(-\ 1)$ is written as -1. We write $\bar{t}$ for the non-empty sequence of terms $t_1, \ldots, t_n$ for an unspecified $n \in \mathbb{N}^+$ that is either irrelevant or clear from the context. Hence, $(\forall \bar{x}.\ t)$ corresponds to a term $(\forall x_1, \ldots, x_n.\ t)$.

In addition to the shorthand characters introduced in Definition 2, we will also use s, t to denote terms, φ, ψ to denote formulas, P to denote a predicate (i.e., a function with codomain sort **Bool**), and c to denote constants.

A substitution is a function that maps variable to terms. Every substitution must map all but finitely many variables to themselves. In general, we will use θ to denote a substitution. The notation $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ denotes the substitution that maps x_i to t_i for $1 \leq i \leq n$ and corresponds to the identity function for all other variables. If θ and η are two substitutions, then $\theta\eta$ denotes the result of first applying θ and then η (i.e., $\eta(\theta(.))$). A substitution can naturally be extended to a function that maps terms to terms by replacing the occurrences of free variables. We write $t[\ ]$ for a term with a hole and $t[u]$ for the term where the hole has been replaced by u. Any term has at most one hole. We also use holes with multiple variables. The notation

$t[\bar{x}_n]$ stands for a term that may depend on distinct variables $\bar{x}_n$. $t[\bar{s}_n]$ is the respective term where the variables $\bar{x}_n$ are simultaneously substituted by $\bar{s}_n$.

Definition 10 (Substitution)
The application of a substitution θ on a term t is defined recursively as

$$\theta(t) = \begin{cases} \theta(x) & \text{if } t \text{ is the variable } x, \\ f(\theta(t_1), \ldots, \theta(t_n)) & \text{if } t = f\ t_1 \ldots t_n, \\ f^\sigma(\theta(t_1), \ldots, \theta(t_n)) & \text{if } t = f^\sigma\ t_1 \ldots t_n, \\ \forall\, x'_1 : \sigma_1 \cdots x'_n : \sigma_n.\ \theta'(t) & \text{if } t = \forall\, x_1 : \sigma_1 \cdots x_n : \sigma_n.\ t, \\ \exists\, x'_1 : \sigma_1 \cdots x'_n : \sigma_n.\ \theta'(t) & \text{if } t = \exists\, x_1 : \sigma_1 \cdots x_n : \sigma_n.\ t, \\ \textsf{let } x'_1 = \theta(t_1) \cdots x'_n = \theta(t_n) \textsf{ in } \theta'(t) & \text{if } t = \textsf{let } x_1 = t_1 \cdots x_n = t_n \textsf{ in } t. \end{cases}$$

To avoid variable capture, bound variables must be renamed if they appear free in a substituted term. Let $F = \{\mathrm{free}(u) \mid u = \theta(x), x \in \mathrm{free}(t), u \neq x\}$.

- The variable x'_i is fresh if $x_i \in F$ and equal to x_i otherwise.
- The substitution θ' is θ, but the variables $x_1, \ldots, x_n$ are mapped to $x'_1, \ldots, x'_n$.

We write $t\theta$ for $\theta(t)$ if it is well sorted.

We omit sorts when they are clear from the context and assume that sort constraints are always respected – that is, substitutions use only terms of the same sort as the substituted variable.

For a set of terms S the set $\mathcal{T}(S)$ is the set of all subterms of the terms in S and $\mathcal{T}(t) = \mathcal{T}(\{t\})$.

Since many-sorted first-order logic does not distinguish between atoms and formulas, some useful concepts from first-order logic have slightly different definitions. The notion of a *trimmed formula* helps to generalize the notion of atom to arbitrary formulas: $\mathrm{trim}(\varphi)$ is the formula φ after removing all leading negations. For example, $\mathrm{trim}(\neg\neg(\varphi_1 \vee \neg\varphi_2))$ is $\varphi_1 \vee \neg\varphi_2$. We say φ is an atom if $\mathrm{trim}(\varphi) = \varphi$. A literal is a formula l, such that either $l = \varphi$ or $l = \neg\varphi$ where φ is an atom. The negation of a literal, denoted $\hat{l}$ is $\neg\varphi$ if $l = \varphi$ and just φ otherwise. Note that the common notation $\bar{l}$ here already denotes a list of literals. A formula is in clause normal form (CNF), if it is of the form

$$(l_{1,1} \vee \cdots \vee l_{n_1,1}) \wedge (l_{1,2} \vee \cdots \vee l_{n_2,2}) \wedge \cdots \wedge (l_{1,m} \vee \cdots \vee l_{n_m,m})$$

where the $l_{i,j}$ are literals. Note that this does not restrict the structure of the literals. They might contain Boolean connectives. Within the context of the SMT solver veriT, this is especially the case for literals starting with a quantifier.

The *polarity* $\mathrm{pol}(\varphi) \in \{+, -\}$ of a formula φ is $-$ (*negative*) if $\mathrm{trim}(\varphi)$ removes an odd number of negations and $+$ (*positive*) otherwise. Furthermore, we borrow the notions of weak and strong quantifiers [7]: since the work presented here is about refuting problems, a positive occurrence of a quantifier $(\exists \bar{x}.\, \varphi)$ or a negative occurrence of a quantifier $(\forall \bar{x}.\, \varphi)$ is strong and a negative occurrence of a quantifier $(\exists \bar{x}.\, \varphi)$ or a positive occurrence of a quantifier $(\forall \bar{x}.\, \varphi)$ is weak. By abuse of notation, we will call the subformula ψ of $(Q\bar{x}.\, \psi)$, where $Q \in \{\forall, \exists\}$, the *matrix*, even though ψ might not be in clausal normal form or even quantifier-free. Without loss of generality, we assume that all quantified variables have been renamed to be distinct.

Definition 11 (Skolemization Operator)
To handle strong quantifiers we use the skolemization operator sko. This operator can only be used in the appropriate context, that is, on strong quantifiers. For a formula $Q\bar{x}.\, \psi$ the operator is defined as

$$\mathrm{sko}(Q\bar{x}.\, \psi, \bar{y}) := \psi[x_1 \mapsto (s_1\ \bar{y}), \ldots, x_n \mapsto (s_n\ \bar{y})]$$

and each s_i is a fresh function symbol of correct sort.
The variables $\bar{y}$ are the variables bound the weak quantifiers that have $Q\bar{x}.\, \psi$ in their scope. If $Q\bar{x}.\, \psi$ is not below a weak quantifier, the list is empty, and the fresh symbols are constants. We write $\mathrm{sko}(Q\bar{x}.\, \psi, \varnothing)$ in this case.

Finally, let us have a look at a proper definition of free variables. It is a simple recursive definition that adds and removes variable as they occur and get bound.

Definition 12 (Free Variables)
For a term t, the set of free variables $\mathrm{free}(t)$ is defined recursively as

$$\mathrm{free}(t) = \begin{cases} \{x\} & \text{if } t \text{ is the variable } x. \\ \mathrm{free}(t_1) \cup \ldots \cup \mathrm{free}(t_n) & \text{if } t = f\ t_1 \ldots t_n, \\ \mathrm{free}(t_1) \cup \ldots \cup \mathrm{free}(t_n) & \text{if } t = f^\sigma\ t_1 \ldots t_n, \\ \mathrm{free}(t) \setminus \{x_1, \ldots, x_n\} & \text{if } t = \forall\, x_1 : \sigma_1 \ldots x_n : \sigma_n.\, t, \\ \mathrm{free}(t) \setminus \{x_1, \ldots, x_n\} & \text{if } t = \exists\, x_1 : \sigma_1 \ldots x_n : \sigma_n.\, t, \\ (\mathrm{free}(t) \setminus \{x_1, \ldots, x_n\}) & \text{if } t = \mathsf{let}\ x_1 = t_1 \ldots x_n = t_n\ \mathsf{in}\ t. \\ \quad \cup \mathrm{free}(t_1) \cup \ldots \cup \mathrm{free}(t_n) & \end{cases}$$

2.2 Solving SMT Problems

In general, any system that solves SMT problems is an SMT solver. This is only half of the story, though. The structure of typical SMT problems goes hand in hand with the most common algorithm used to solve them. The core idea is to combine propositional reasoning with theory reasoning. The standard algorithm to propositional reasoning is *conflict-driven clause learning* (CDCL). Extended to theories, this becomes CDCL(T). The T stands for "Theories", but can also be replaced by the name of a concrete theory. There are two ways to look at CDCL(T): either as a procedural algorithm or as an abstract calculus. Since the work presented here touches on preprocessing and other subjects related to the concrete implementation, we will focus on the procedural description. The abstract calculus also gets some attention at the end of this section. A good starting point to learn more about SMT solving is the corresponding chapter in the *Handbook of Satisfiability* [18].

The basic idea of CDCL(T) is to separate propositional reasoning from theory reasoning by creating an *abstract* version of the problem where every literal is replaced by a propositional variable. Standard CDCL determines which propositional variables, and hence which literals, must hold. A theory solver then determines if those literals are also valid in the theory. If this is the case the problem is solved, if it is not, then new clauses are generated. Should the problem contain quantifiers, then an additional component instantiates the quantifiers.

This approach to SMT solving is known as the *lazy* approach. The eager approach, on the other hand, works by encoding the input problem into a propositional satisfiability problem up front. The translation must preserve the satisfiability of the problem. The propositional problem can then be handed over to any standard SAT solver.

Let us now look at the lazy approach in more detail. The following description is inspired by the CDCL(T) implementation in veriT, but is greatly simplified.

Figure 2.1 shows the operation of a CDCL(T) based SMT solver when solving a problem with quantifiers. It first preprocesses the input problem. Then two procedures together refute the problem: the ground solver either refutes the problem at the ground level, or finds a ground model. For problems without quantifiers the process would terminate now, but if quantifiers are present the instantiation procedure is started with the ground model.

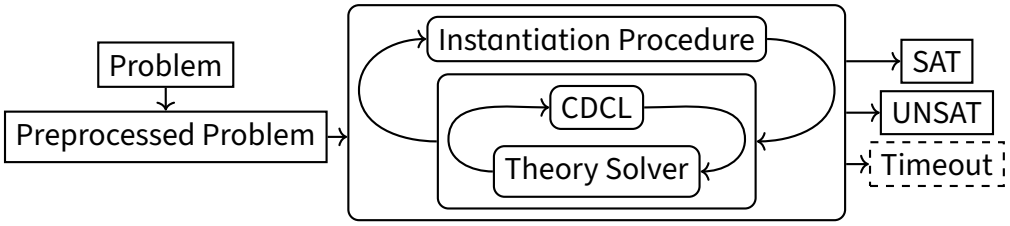


Figure 2.1 The main loop of an SMT solver.

The procedure either creates new ground lemmas or can determine that the problem is satisfiable (Section 2.2.2).

Given an input sentence (i.e., a closed term of sort **Bool**) an SMT solver performs multiple preprocessing steps before the solving phase is started. This produces an equisatisfiable problem F in clause normal form.

To make efficient use of the ground solver, ground subformulas are fully clausified. Quantified formulas, however, are treated differently. They are usually not put in prenex form or clausified. Furthermore, strong quantifiers are usually not fully skolemized. This has the benefit that the original structure of quantified formulas is preserved, which is crucial for some instantiation techniques.

Preprocessing applies some light form of rewriting on quantified formulas. In veriT, most rewriting steps apply to constants below arithmetic operators and Boolean connectives. For example, the term $(f\ (5 + c_1 + 3)\ (c_2 \times (3 - 3)))$ is simplified to $(f\ (8 + c_1)\ 0)$ and $(\bot \to \varphi_1) \to \varphi_2$ is replaced by φ_2. Rewriting also ensures that certain global invariants of veriT are met: for instance, all occurrences of bound variables are renamed to distinct variables, and quantifiers over Boolean variables are removed by Shannon expansion.

Skolemization is another preprocessing step applied to quantified formulas. How skolemization is applied is essentially implementation dependent. The common abstract CDCL(T) calculus [18] is only concerned with ground reasoning.

While the SMT solver Z3 [47] has a builtin tactic called *nnf* that fully applies skolemization, cvc5 [11] and veriT only skolemize outermost strong quantifiers in their default configuration. The rewriter of veriT skolemizes outermost strong quantifiers by replacing the subformula $\mathrm{trim}(l)$ of a formula l by $\mathrm{sko}(\mathrm{trim}(l), \varnothing)$ if $\mathrm{trim}(l)$ has the form $Q\bar{x}.\ \varphi$, the quantifier Q is strong, and l does not occur below any quantifier. Furthermore, veriT also replaces $\neg(\exists \bar{x}.\ \varphi)$ with $(\forall \bar{x}.\ \neg\varphi)$. Most SMT solvers handling quantified

formulas apply such light rewriting steps, but they do not apply powerful reasoning on quantified formulas.

Due to the limited preprocessing of quantified formulas, some literals of F start with a weak quantifier and contain complicated formulas. The ground solver cannot work with such formulas. To clarify the distinction, we call the disjuncts which start with a quantifier, and are hence black boxes to the ground solver, *boxes*. In this thesis we assume that all boxes are universally quantified. This is assured by the preprocessing of veriT. Disjuncts that are not boxes are ground literals. A *unit-box* is a clause with only one disjunct that is a box.

Example 4
We can illustrate boxes by replacing quantified formulas by numbered frames. Consider the clauses $(\forall x.\, P\, x)$, $\neg(P\, a) \vee (\forall x.\, P\, x)$. $(P\, a) \vee (\forall x.\, P\, x) \vee (\forall y.\, \bot)$, Then illustrated with boxes, the problem becomes $\boxed{1}$, $\neg(P\, a) \vee \boxed{1}$, $(P\, a) \vee \boxed{1} \vee \boxed{2}$. The original problem is unsatisfiable, but the ground solver can generate a ground model for the abstracted problem. It cannot see the constant $\bot$ in $\boxed{2}$.

Extensive preprocessing is necessary to solve many problems, and solvers implement a wide palette of techniques. This provides a challenge if the user demands a proof for the preprocessing steps. Since techniques are often substantially different, expressing proofs for all of them within one proof framework is difficult.

The solving loop in Figure 2.1 starts with the *ground solver* searching for a ground model of the preprocessed problem F. To do so, it first generates a propositional abstraction of the problem. Every literal l in F is mapped to a propositional literal l^a such that $\hat{l}^a = \widehat{l^a}$ and if $l_1^a = l_2^a$ then $l_1 = l_2$.

Since we assume that after preprocessing quantified formulas never start with a negation, boxes are always assigned to a positive literal. This assignment is not an explicit step in veriT. Instead, it uses a data structure to represent terms that assigns a unique number to each syntactically distinct term. This number is directly used to represent propositional variables too. Overall, the result of this operation is a propositional problem F^a of the form

$$(l^a_{1,1} \vee \ldots \vee l^a_{n_1,1}) \wedge (l^a_{1,2} \vee \ldots \vee l^a_{n_2,2}) \wedge \ldots \wedge (l^a_{1,m} \vee \ldots \vee l^a_{n_m,m}).$$

In the following we discuss the inner loop of the SMT solver as shown in Figure 2.1. It tries to find a model of the quantifier-free part of the problem. First, the CDCL algorithm searches for a propositional model of the abstracted problem. Then, the theory solvers checks if the literals in the propositional models are consistent in the theories. If this is the case, the literals form the ground model.

Definition 13 (Propositional Model)
Let $\mathscr{M}^a$ be a subset of $\{l^a_{1,1}, \ldots, l^a_{n_m,m}\}$ such that if $l \in \mathscr{M}^a$, then $\hat{l} \notin \mathscr{M}^a$. The set $\mathscr{M}^a$ is a propositional model of F if the propositional abstraction F^a is satisfied if its literals that are in $\mathscr{M}^a$ are set to true.

If the SMT solver determines that no propositional model exists (i.e., the abstraction is propositionally unsatisfiable), the original problem is unsatisfiable and the SMT solver terminates. If, on the other hand, a propositional model is found, the theory solvers jump into action. The theory solvers depend on the theory used, and their calculus can vary widely. Furthermore, if more than on theory is used, it is necessary to combine the theory solvers for each theory. In any case, the theory solver takes as input the propositional model and determines either that the model is consistent with the theories or that it contradicts them. If the literals of the propositional model are consistent with the theories, then they form a ground model of the quantifier-free part of F. Since quantifier reasoning is handled by the dedicated instantiation module, such a model is usually just called "ground model" in SMT parlance.

Definition 14 (Ground Model)
Let $\mathscr{M}^a = \{l^a_1, \ldots, l^a_n\}$ be a propositional model of F^a. The set $\mathscr{M} = \{l_1, \ldots, l_n\}$ is a ground model of F if $\vDash_T \bigwedge_{l \in \mathscr{M} \setminus B} l$ where B is the set of literals in $\mathscr{M}$ that contain boxes. That is, $\mathscr{M}$ is a ground model if the literals that do not contain quantifiers are valid in the considered theory.

Example 5
Consider the problem $F = \{(Q\ a), \neg(Q\ a) \vee a \approx b, (P\ a) \not\approx (P\ b) \vee \forall x.\, \neg(Q\ x)\}$ in the Core theory. For the ground solver, this problem looks like $F = \{(Q\ a), \neg(Q\ a) \vee a \approx b, (P\ a) \not\approx (P\ b) \vee \boxed{1}\}$.
The set of literals $\{(Q\ a), a \approx b, (P\ a) \not\approx (P\ b)\}$ is one possible propositional model. The theory solver, however, will detect that $a \approx b$ contradicts $(P\ a) \not\approx (P\ b)$ and will add the new clause $\neg(Q\ a) \vee a \not\approx b \vee (P\ a) \approx (P\ b)$ to F.

Subsequently, the ground solver will produce the ground model $\mathcal{M} = \{(Q\ a), a \approx b, \boxed{1}\}$. It is now the task of the instantiation procedure to generate an instance of $(P\ a) \not\approx (P\ b) \vee \forall x.\, \neg(Q\ x)$.

If the propositional model is a ground model, the solver either continues with quantifier instantiation, or, if no quantified formulas are present, it can conclude that the problem is satisfiable. If the propositional model is not a ground model, the theory solver must hand the control back to the propositional solver. Since the propositional solver should generate a different model, the theory solver must extend F with new clauses. The simplest way to do so is to first negate the literals in $\mathcal{M}^a$ to obtain a new set $\hat{\mathcal{M}}^a$ and to then form a new clause by combining the literals from $\hat{\mathcal{M}}^a$ conjunctively. When this new clause is added to F, it ensures that all subsequent models assign a different value to at least one of its literals.

Of course, a theory solver could also add clauses generated by a more sophisticated method. For example, it could reduce the number of literals by searching for an unsatisfiable core. Overall, many optimizations exist for this process. Usually, the CDCL process is tightly coupled to the theory solvers. Instead of building a full propositional model, it communicates every decision and unit propagation directly to the theory solvers. The theory solvers can detect early on when the current partial model becomes inconsistent in the theories and can generate an appropriate clause. Many theory solvers can also be backtracked to a decision point to avoid complete resets of the theory solvers. Nevertheless, the simplified presentation of the ground solver we just saw is enough to understand the rest of this thesis.

The last remaining piece of a full SMT solver is the quantifier instantiation procedure. Should the ground solver determine that the propositional model is a ground model, control is handed over to the instantiation procedure. This procedure searches for ground substitutions for quantified formulas appearing in F. More precisely, let $C[\forall x_1, \ldots, x_n.\, \varphi]$ be a clause in F. Then the instantiation procedure searches for a substitution θ that maps the variables $x_1, \ldots, x_n$ to variable-free terms. If such a substitution is found, the procedure derives $C[\forall x_1, \ldots, x_n.\, \varphi] \to C[\varphi\theta]$. Usually, more than one quantifier is instantiated. In the best case, all quantifiers in the clause are instantiated, and a ground clause is obtained. In any case, the results are added to F and control is handed back to the ground solver. Multiple methods exist to generate the substitutions. We discuss them in Section 2.2.2, but we will first have a look at the abstract calculus for SMT solving.

2.2.1 The Abstract CDCL(T) Calculus

The description here is based on a presentation given by Cesare Tinelli at the SC^2 summer school in 2017[7] on the common abstract CDCL(T) calculus [77].

The abstract calculus is a rewrite system on states. A state is either a tuple $\langle M, F \rangle$, or a special **fail** state. The first component of the tuple M is a sequence of literals and decision points $\bullet$. This sequence is called the trace and represents the partial ground model during solving. The second component F is the set of clauses as before that represents the problem as it changes by the ongoing solving process. Let $M = M_0 \bullet M_1 \bullet \cdots \bullet M_n$ such that no M_i contains a decision point. Then M_i is at *decision level* i of M and we write $M[i]$ for $M_0 \bullet \cdots \bullet M_i$. The process starts with an initial state $\langle\ , F_0 \rangle$ where F_0 is the problem in clause normal form just after preprocessing. Then the states are transformed by a set of rules that have the form

$$\frac{p_1 \quad \cdots \quad p_n}{[M \leftarrow e_1] \quad [F \leftarrow e_2]}$$

where p_1 to p_n are premises and M, F, or both are updated.

The abstract SMT solver applies rules until an end state is reached. For example, a simple DPLL procedure uses four rules: PROPAGATE, DECIDE, FAIL, and BACKTRACK. The rules treat clauses modulo the associativity and commutativity of disjunction. The dedicated **fail** state indicates that the initial state is unsatisfiable. This state is reached by a dedicated rule.

$$\frac{l_1 \vee \ldots \vee l_n \in F \quad \hat{l}_1, \ldots, \hat{l}_n \in M \quad \bullet \notin M}{\textbf{fail}}\ \text{FAIL}$$

If at any point the trace negates all literals on a clause and there is no decision point to backtrack to, then the problem is propositionally unsatisfiable and the solver is done.

Decisions taken by the SAT solver are also modeled by a dedicated rule. It introduces a new decision point.

$$\frac{l \text{ is a literal that appears in } F, \text{ or its negation.} \quad l, \hat{l} \notin M}{M \leftarrow M \bullet l}\ \text{DECIDE}$$

Whenever only one literal in a clause is not assigned a value, this literal must be true. This information can then be added to the trace.

[7] The slides are available at http://www.sc-square.org/CSA/school/lectures/SCSC-Tinelli.pdf.

$$\frac{l_1 \vee \ldots \vee l_n \vee l \in F \quad \hat{l}_1, \ldots, \hat{l}_n \in M \quad l, \hat{l} \notin M}{M \leftarrow M\, l} \text{ PROPAGATE}$$

If the trace leads to a contradiction, like in FAIL, but the trace contains a decision point, then the solver can backtrack. Since we model simple DPLL, BACKTRACK enforces chronological backtracking with the last constraint.

$$\frac{l_1 \vee \ldots \vee l_n \in F \quad \hat{l}_1, \ldots, \hat{l}_n \in M \quad M = N \bullet l\, O \quad \bullet \notin O}{M \leftarrow N\, \hat{l}} \text{ BACKTRACK}$$

The propositional solving process terminates when a state $\langle M, F \rangle$ is found where the model $\mathscr{M}$ obtained by removing the decision points from M satisfies F. The set of rules can be extended and modified to model more advanced calculi, such as CDCL. To model clause learning, an additional conflict clause must be added to the state.

Example 6
The propositional reasoning in Example 5 corresponds to the following transformations. Here $F_0 = \{(Q\ a), \neg(Q\ a) \vee a \approx b, (P\ a) \not\approx (P\ b) \vee \forall x.\, \neg(Q x)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{F \leftarrow F_0 \quad M \leftarrow}{F \leftarrow F_0 \quad M \leftarrow (Q\ a)} \text{ PROPAGATE}}{F \leftarrow F_0 \quad M \leftarrow (Q\ a)\ a \approx b} \text{ PROPAGATE}}{F \leftarrow F_0 \quad M \leftarrow (Q\ a)\ a \approx b \bullet (P\ a) \not\approx (P\ b)} \text{ DECIDE}$$

Beyond propositional reasoning, additional rules are needed to model reasoning with theories. The very simple integration of a theory solver with the ground solver as discussed before needs only one rule.

$$\frac{l_1, \ldots, l_n \in M \quad l_1, \ldots, l_n \vDash_T \bot}{F \leftarrow F \cup \{\hat{l}_1 \vee \ldots \vee \hat{l}_n\}} \text{ T-CONFLICT}$$

With this rule, the ground SMT solver finishes when either the state **fail** is reached, or the trace is a propositional model as before and the rule T-CONFLICT cannot be applied.

Example 7
Continuing from Example 6 the SMT solver performs the steps

$$\frac{F \leftarrow F_0 \quad M \leftarrow \cdots \bullet (P\ a) \not\approx (P\ b)}{F \leftarrow F_0 \cup \{\neg(Q\ a) \vee a \not\approx b \vee (P\ a) \approx (P\ b)\}}\ \text{T-CONFLICT}$$

$$\frac{M \leftarrow \cdots \bullet (P\ a) \not\approx (P\ b)}{M \leftarrow \cdots\ (P\ a) \approx (P\ b)}\ \text{BACKTRACK}$$

Yet another rule can model quantifier instantiation. In our case, the condition on when this rule can be applied is very strict: the ground SMT solver must have finished and not be in the **fail** state.

$$\frac{C[\forall x_1, \ldots, x_n.\varphi] \in F \quad \theta \text{ is a substitution built using } M}{F \leftarrow F \cup \{C[\varphi\theta]\}}\ \text{INSTANTIATE}$$

Example 8
The SMT solver uses quantifier instantiation to continue from Example 7. Here, $F_1 = F_0 \cup \{\neg(Q\ a) \vee a \not\approx b \vee (P\ a) \approx (P\ b)\}$. The clause instantiated is $(P\ b) \vee \forall x.\, \neg(Qx)$ and the instantiation used is $[x \mapsto a]$. In the next section, we will see how instantiation procedures actually find this instance.

$$\frac{F \leftarrow F_1 \quad M \leftarrow (Q\ a)\ a \approx b\ (P\ a) \approx (P\ b)}{F \leftarrow F_1 \cup \{(P\ a) \not\approx (P\ b) \vee \neg(Qa)\}}\ \text{INSTANTIATE}$$

$$\frac{M \leftarrow (Q\ a)\ a \approx b\ (P\ a) \approx (P\ b)}{\textbf{fail}}\ \text{FAIL}$$

2.2.2 Quantifier Instantiation Techniques

Let us now open the box containing the quantifier instantiation techniques. Those have to use the current ground model and clause set to generate new instances of the quantified formulas. The goal is to generate instances that lead the ground solver to conclude that the problem is unsatisfiable. Furthermore, the instantiation techniques also have the chance to detect that the problem has a first-order model and is satisfiable.

Overall, there are four common techniques for quantifier instantiation that we will discuss here. Conflict-driven instantiation searches for instances that contradict the current ground model. Trigger-based instantiation generates instances by matching patterns. Enumerative instantiation generates all possible instances one after another. Finally, model-based instantiation extends the ground model to a first-order model either to determine that

the problem is satisfiable, or to generate an instance that expresses a contradiction with the first-order model. The veriT solver implements all of these instantiation techniques, except model-based instantiation. To combine them into one instantiation procedure, it simply tries the methods one by one in the order they are presented here. It stops at the first technique that successfully generates new instances.

The four instantiation techniques that we present here can be expressed as a E-ground (dis)unification problem [10, 13]. This has the potential to simplify the implementation, since the different instantiation methods can share the code to solve the E-ground (dis)unification problem. The goal of this problem is to find a substitution θ, such that

$$e_1 \wedge \ldots \wedge e_n \vDash_E (l_1 \wedge \ldots \wedge l_m)\theta$$

where $e_1, \ldots, e_n, l_1, \ldots, l_m$ are equality literals (i.e., literals l such that $\mathrm{trim}(l)$ starts with an equality), the literals e_i are ground literals, and some literals l_i contain free variables. Note that non-equality literals l can be translated to equality literals. If $l = \varphi$ replace it by $\varphi \approx \top$. If $l = \neg\varphi$ replace it by $\varphi \approx \bot$. The solution substitution θ is used to generate the instance. Usually $e_1, \ldots, e_n$ are the literals in the ground model. Correspondingly, we will often write E-ground (dis)unification problems as $\mathscr{M} \vDash_{\text{Core}} \ldots$. Solving the E-ground (dis)unification problem is NP-hard. It can be solved by the congruence closure with free variables (CCFV) algorithm. This algorithm interprets the E-ground (dis)unification problem as a constraint system that it disassembles to iteratively build the substitution. If it can deduce the assignment for an individual variable, the substitution is extended with this assignment and applied to the remaining constraints. The CCFV algorithm is implemented in veriT and used for conflict-driven and trigger-based instantiation. It is efficient in practice.

Conflict-Driven Instantiation. The core idea behind conflict-driven instantiation is to generate instances that contradict the current ground model. Hence, this technique somewhat resembles a theory solver. It is a recent addition to the zoo of instantiation techniques [4]. Conflict-driven instantiation is very helpful, since it only generates instances that are immediately useful. It forces the ground solver to find new models and eliminates spurious models from the search space.

This method tries for each clause $l_1 \vee \ldots \vee l_n \in F$ that contains a box to find a substitution such that every literal of the clause is contradicted by the

current ground model in the theory of equality and uninterpreted functions. This can be achieved by solving the E-ground (dis)unification problem

$$\mathcal{M} \models_E (\hat{l}'_1 \wedge \ldots \wedge \hat{l}'_m)\theta$$

where each literal l'_1 is l_1 if l_1 does not start with a quantifier. Otherwise, l'_1 is φ if l_1 has the form $\forall \bar{x}.\, \varphi$.

To achieve the goal of conflict-driven instantiation, it is enough to solve this problem for one clause. Nevertheless, it is useful to generate as many such instances as possible to restrict the search space.

Example 9
In Example 8 conflict-driven instantiation must find an instance of $\neg(Q\ x)$ that contradicts the model. Since $(Q\ a)$ is part of the ground model, $[x \mapsto a]$ is an easy solution. The substitution $[x \mapsto b]$ is also a solution: since $a \approx b$ is part of the ground model it also contradicts the model in the Core theory.

Trigger-Based Instantiation. This instantiation scheme works by matching *triggers* with the current ground model. It is a classic method used by SMT solvers to handle quantifiers and is used by the solver Simplify [40]. Triggers associate with every box $(\forall \bar{x}.\, \varphi)$ one or more lists of quantifier-free terms $t_1, \ldots, t_n$ such that $\mathrm{free}(t_1) \cup \ldots \cup \mathrm{free}(t_n) = \{\bar{x}\}$. The process of matching terms modulo a set of ground equalities over uninterpreted function symbols is called *E-matching* [40, 45, 56]. To construct instances of φ, E-matching searches for substitutions θ and terms $g_1, \ldots, g_n \in \mathcal{T}(\mathcal{M})$ such that

$$\mathcal{M} \models_{\mathrm{Core}} (t_1 \approx g_1 \wedge \ldots \wedge t_n \approx g_n)\theta$$

If the search is successful, it returns the instance $\varphi\theta$. This can be modeled as an E-ground (dis)unification problem by using fresh variables $y_1, \ldots, y_n$ and the constraint

$$\mathcal{M} \models_{\mathrm{Core}} (t_1 \approx y_1 \wedge \ldots \wedge t_n \approx y_n)\theta.$$

Furthermore, the solution θ must map the variables $y_1, \ldots, y_n$ to terms in $\mathcal{T}(F)$.

The triggers are either provided by the user as annotations to the problem or are heuristically generated during preprocessing. Trigger generation uses the structure of the quantified formulas, which is preserved by preprocessing. Every subterm of the quantified formula is a potential trigger, and heuristics

are used to select promising subterms [68]. Care must be taken to avoid an effect called "matching loop".

Example 10
Consider the formula $\forall x.\,(f\ (f\ x)) \approx (f\ x)$ with the trigger $(f\ x)$. The substitution $[x \mapsto t]$ generates the instance $(f\ (f\ t)) \approx (f\ t)$ that adds the term $(f\ (f\ t))$ to the problem. Later, this leads to the substitution $[x \mapsto (f\ t)]$ that adds $(f\ (f\ (f\ t)))$ and so on [79].

Matching loops lead to instances that are only used to generate further instances, which leads to an explosion of the size of the ground problem. They can be avoided either by being mindful when generating triggers [68] or by extending the instantiation procedure to detect instances that are not used by the ground solver [45].

A unique aspect of trigger-based instantiation is that it can be controlled by the user through annotations. For example, this can be used to express domain knowledge about instances that should be generated. Furthermore, it is also possible to deactivate all other instantiation techniques and restrict trigger-based instantiation to user provided triggers. The result is a programmable SMT solver, and it is possible to use this to build complete solvers for new theories [43].

Due to the heuristic nature of automatically inferred triggers, the generated instances might not be useful to solve the problem. Instead, they can slow down or mislead the solver.

Example 11
If the instantiation procedure uses trigger-based instantiation to continue Example 8, it would produce the instances $\neg(Q\ a)$ if the trigger $(Q\ x)$ is used. This trigger matches the term $(Q\ a)$ in the model.

Enumerative Instantiation. While conflict-driven instantiation is guided by the ground model it tries to contradict, and trigger-based instantiation is guided by the triggers, enumerative instantiation [80] is unguided. For a box with the form $(\forall \bar{x}.\,\varphi)$ it creates all substitutions $[\bar{x} \mapsto \bar{t}]$ where the terms $\bar{t}$ are ground terms from $\mathcal{T}(F)$. To limit the number of generated instances, the procedure only uses the ground terms minimal with respect to some term order and does not return instances already implied by the ground model (i.e., it only returns $\varphi\theta$ if $\mathcal{M} \not\models_{\text{Core}} \varphi\theta$).

Enumerative instantiation can also ensure that the SMT solver is complete for the Core theory. To ensure completeness, the instances must be generated in a fair manner [51,64].

A special case is the case where enumerative instantiation fails to generate new instances. If the problem uses only the theory of uninterpreted functions, the SMT solver can conclude that it is satisfiable. This, however, only works if the solver can guarantee that no instance is missed due to errors or intentional approximations in the data structures used by the instantiation procedure. Hence, veriT will not state that a problem is satisfiable if enumerative instantiation does not generate any instance.

Due to its unguided nature enumerative instantiation tends to generate many useless instances. Within the mix of instantiation techniques, however, it can find the small ground terms that are sometimes necessary to enable the two previous techniques to work. Hence, it is a useful fallback strategy.

Example 12
For Example 8, enumerative instantiation would produce the two instances $\neg(Q\ a)$ and $\neg(Q\ b)$, because a and b are ground terms occurring in F.

Model-Based Quantifier Instantiation. This method extends heuristically the ground model $\mathscr{M}$ to a first-order interpretation $\mathscr{I}$ and tests if this interpretation is a model of the full problem [57].

The construction of the candidate interpretation $\mathscr{I}$ is very simple. It uses if-then-else constructs for function symbols. Their interpretation corresponds to the assignments implied by $\mathscr{M}$ on known terms. All other values are mapped to a default value.

The candidate interpretation $\mathscr{I}$ is not a model of F if there exists a clause $l_1 \vee \ldots \vee l_n \in F$ and a substitution θ of $\mathrm{free}(l_1 \vee \ldots \vee l_n)$ with terms from $\mathscr{T}(F)$ such that $\mathscr{I} \models \neg(l_1 \vee \ldots \vee l_n)\theta$. This test can be expressed as an E-ground (dis)unification problem. If no clause and substitution that contradict the interpretation can be found, then $\mathscr{I}$ is a model of F and the input problem is satisfiable. If, on the other hand, such a clause and substitution exists, then the ground instance $(l_1 \vee \ldots \vee l_n)\theta$ is produced. This will ensure that subsequent ground models lead to different interpretations. This method is complete for many-sorted first-order logic and several fragments of first-order logic modulo theories [57].

Model-based quantifier instantiation is especially useful to detect the satisfiability of the input problem. It is generally assumed that the technique

is less useful to generate instances that eventually allow the SMT solver to show unsatisfiability. Hence, it is not implemented in veriT.

Example 13
Again, for the last time, let us have a look at Example 8. The candidate interpretation would interpret the function P with a constant function that maps every input to a default value. The predicate Q, on the other hand, must be mapped to $\top$ for both the argument a and b. The reason for this is the presence of both $(Q\ a)$ and $a \approx b$ in the ground model. This forces $(Q\ a)$ to $\top$ Since a is equal to b in this model, $(Q\ b)$ is also mapped to $\top$.
We can use the clause $(P\ a) \not\approx (P\ b) \vee \forall x. \neg(Q\ x) \in F$ to show that this candidate interpretation is not a model. If we replace x by a, we get $(P\ a) \not\approx (P\ b) \vee \neg(Q\ a)$. The candidate interpretation interprets both literals in this clause with $\bot$.

III Improving Proofs

If an SMT solver determines that an input problem is satisfiable, it might provide a model to the user. If it determines that the problem is unsatisfiable, it might provide a proof that exhibits the unsatisfiability of the problem. When an SMT solver is used by Sledgehammer or by the `smt` tactic, we are in the second scenario. Both tools *negate* the current proof obligation, add the selected background lemmas, and encode the result as an SMT-LIB problem. Hence, the SMT solver is successful if it can show the validity of the proof obligation and selected lemmas by showing the unsatisfiability of the generated problem. While Sledgehammer is content with an unsatisfiability core during the filtering phase, the `smt` tactic needs a proof that it can reconstruct within the trusted kernel of Isabelle/HOL.

This chapter discusses a proof format for SMT solvers and its reconstruction by the `smt` tactic. The format is called "Alethe" and it builds upon earlier formats used by the solver veriT. Proof production from veriT goes back ten years [20]. Since then, it was refined and extended. Most notably, the proof format was extended with a notion of context to express reasoning with binders [12].

The work on implementing the proof reconstruction for the `smt` tactic exposed shortcoming with the proofs generated by veriT. First, there was no complete documentation and specification of the proof rules available. Section 3.1 fills this gap – it is a full documentation of the Alethe proof format. Section 3.1 also clarifies the connection between abstract proofs and the concrete proofs produced by veriT. Second, some proof rules were hard to reconstruct and there were gaps in the proofs. In Section 3.2 we discuss improvements to the format that solve these issues. We also used this effort to overhaul the concrete syntax of Alethe proofs. They now use a SMT-LIB-based syntax that uses a flat list of commands instead of a nested structure.

Overall, Section 3.2 discusses the reconstruction of Alethe proofs implemented by the `smt` tactic of Isabelle/HOL. This section also discusses the aforementioned improvements to veriT's proofs. The empirical evaluation shows that the addition of veriT support to `smt` reduces the number of preplay failures of the Sledgehammer pipeline.

Overall, the implementation of the proof reconstruction was successful, because we also improved the proof format. Accordingly, Section 3.3 we

discuss both opportunities to improve the reconstruction and the Alethe proof format.

3.1 The Alethe Proof Format

This section is based on an informally published reference of the Alethe proof format. It was originally written by me for the work in Section 3.2. Later Haniel Barbosa, Mathias Fleury, Pascal Fontaine contributed.

This section is a reference of the Alethe[8] proof format. Alethe is designed to be a flexible format to represent unsatisfiability proofs generated by SMT solvers. Alethe proofs can be consumed by other systems, such as interactive theorem provers or proof checkers. The design is based on natural-deduction style structure and rules generating and operating on first-order clauses. The Alethe calculus consists of two parts: the proof language based on SMT-LIB and a collection of proof rules. Section 3.1.1 introduces the language. First as an abstract language, then as a concrete syntax. Section 3.1.2 discusses the core concepts behind the Alethe proof rules. The Appendix "The Alethe Proof Rules" presents a list of all proof rules currently used by veriT.

Alethe follows a few core design principles. First, proofs should be easy to understand by humans to ensure working with Alethe proofs is easy. Second, the language of the format should directly correspond to the language used by the solver. Since many solvers use the SMT-LIB language, Alethe also uses this language. Therefore, Alethe's base logic is the many-sorted first-order logic of SMT-LIB. Third, the format should be uniform for all theories used by SMT solvers. With the exception of clauses for propositional reasoning, there is no dedicated syntax for any theory.

The format is currently used by the SMT solver veriT. If requested by the user, veriT outputs a proof if it can deduce that the input problem is unsatisfiable. In proof production mode, veriT supports the theory of uninterpreted functions, the theory of linear integer and real arithmetic, and quantifiers. The Alethe proofs can be reconstructed by the `smt` tactic of the proof assistant Isabelle/HOL [54,88] (see Section 3.2). The SMTCoq tool can reconstruct an older version of the format in the proof assistant Coq [52]. An effort to update the tool to the latest version of Alethe is ongoing. The SMT solver cvc5 [11] (the successor of CVC4) supports Alethe experimentally as one of its multiple proof output formats. Furthermore, there is an experimental high-performance proof checker written in Rust.[9]

[8] Alethe is a genus of small birds that occur in West Africa [99]. The name was chosen, because it resembles the Greek word αλήθεια (alítheia) – truth.

[9] Available at https://github.com/ufmg-smite/alethe-proof-checker.

In addition to this reference, the proof format has been discussed in past publications, which provide valuable background information. The core of the format goes back to 2011 when two publications at the PxTP workshop outlined the fundamental ideas behind the format [20] and proposed rules for quantifier instantiation [38]. More recently, the format has gained support for reasoning typically used for processing, such as skolemization, substitutions, and other manipulations of bound variables [12].

3.1.1 The Alethe Language

This section provides an overview of the core concepts of the Alethe language and also introduces some notation used throughout this chapter. The section first introduces an abstract notation to write Alethe proofs. Then, it introduces the concrete, SMT-LIB-based syntax. Finally, we show how a concrete Alethe proof can be checked.

Example 14
The following example shows a simple Alethe proof expressed in the abstract notation used in this document. It uses quantifier instantiation and resolution to show a contradiction. The paragraphs below describe the concepts necessary to understand the proof step by step.

1.	▷	$\forall x.\,(P\ x)$	assume
2.	▷	$\neg(P\ a)$	assume
3.	▷	$\neg(\forall x.\,(P\ x)) \vee (P\ a)$	forall_inst $[(x, a)]$
4.	▷	$\neg(\forall x.\,(P\ x)), (P\ a)$	(or 3)
5.	▷	$\bot$	(resolution $1, 2, 4$)

Many-Sorted First-Order Logic. Alethe builds on the SMT-LIB language. This includes its many-sorted first-order logic described in Section 2.1. The available sorts depend on the selected SMT-LIB theory/logic as well as on those defined by the user, but the distinguished **Bool** sort is always available. However, Alethe also extends this logic with Hilbert's choice operator ε. The term $\varepsilon x.\,\varphi[x]$ stands for a value v such that $\varphi[v]$ is true if such a value exists. Any value is possible otherwise. Alethe requires that ε is functional with respect to logical equivalence: if for two formulas φ, ψ that contain the free variable x, it holds that $(\forall x.\,\varphi \approx \psi)$, then $(\varepsilon x.\,\varphi) \approx (\varepsilon x.\,\psi)$ must also hold.

Note that, choice terms can only appear in Alethe proofs, not in SMT-LIB problems.

Steps. A proof in the Alethe language is an indexed list of steps. To mimic the concrete syntax of Alethe proofs, proof steps in the abstract notation have the form

$$i.\ c_1,\ \ldots,\ c_j\ \triangleright \qquad l_1, \ldots, l_k \qquad (\mathsf{rule}\ p_1,\ \ldots,\ p_n)\ [a_1,\ \ldots,\ a_m]$$

Each step has a unique index $i \in \mathbb{I}$, where $\mathbb{I}$ is a countable infinite set of valid indices. In the concrete syntax all SMT-LIB symbols are valid indices, but for examples we will use natural numbers. Furthermore, $l_1, \ldots, l_k$ is a clause with the literals l_i. It is the conclusion of the step. If a step has the empty clause as its conclusion (i.e., $k = 0$) we write $\bot$. While this muddles the water a bit with regard to steps which have the unit clause with the unit literal $\bot$ as their conclusion, it simplifies the notation. We will remark on the difference if it is relevant. The rule name rule is taken from a set of possible proof rules (see the Appendix "The Alethe Proof Rules"). Furthermore, each step has a possibly empty set of premises $\{p_1, \ldots, p_n\} \subseteq \mathbb{I}$, and a rule-dependent and possibly empty list of arguments $[a_1, \ldots, a_m]$. The list of premises only references earlier steps, such that the proof forms a directed acyclic graph. If the list of premises is empty, we will drop the parentheses around the proof rule. The arguments a_i are either terms or tuples (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The list $c_1, \ldots, c_j$ is the *context* of the step. Contexts are discussed below. Every proof ends with a step that has the empty clause as the conclusion and an empty context. The list of proof rules in the appendix also uses this notation to define the proof rules.

The example above consists of five steps. Step 4 and 5 use premises. Since step 3 introduces a tautology, it uses no premises. However, it uses arguments to express the substitution $[x \mapsto a]$ used to instantiate the quantifier. Step 4 translates the disjunction into a clause. In the example above, the contexts are all empty.

Assumptions. An assume step introduces a term as an assumption. The proof starts with a number of assume steps. Each such step corresponds to an input assertion. Within a subproof, additional assumptions can be introduced too. In this case, each assumption must be discharged with an appropriate step. The rule subproof can be used to do so. In the concrete

syntax, assume steps have a dedicated command `assume` to clearly distinguish them from normal steps that use the `step` command (see Section 3.1.1.1).

The example above uses two assumptions which are introduced in the first two steps.

Subproofs and Lemmas. Alethe uses subproofs to prove lemmas and to create and manipulate the context. To prove lemmas, a subproof can introduce local assumptions. The subproof *rule* discharges the local assumptions. From an assumption φ and a formula ψ proved from φ, the subproof rule deduces the clause $\neg\varphi, \psi$ that discharges the local assumption φ. A subproof step cannot use a premise from a subproof nested within the current subproof.

Subproofs are also used to manipulate the context. As the example below shows, the abstract notation denotes subproofs by a frame around the steps in the subproof. In this case the subproof concludes with a step that does not use the subproof rule, but another rule, such as the bind rule.

Example 15
This example shows a refutation of the formula $(2 + 2) \approx 5$. The proof uses a subproof to prove the lemma $((2 + 2) \approx 5) \Rightarrow 4 \approx 5$.

1.	▷	$(2 + 2) \approx 5$	assume
2.	▷	$(2 + 2) \approx 5$	assume
3.	▷	$(2 + 2) \approx 4$	sum_simplify
4.	▷	$4 \approx 5$	(trans 2, 3)
5.	▷	$\neg((2 + 2) \approx 5), 4 \approx 5$	subproof
6.	▷	$(4 \approx 5) \approx \bot$	eq_simplify
7.	▷	$\neg((4 \approx 5) \approx \bot), \neg(4 \approx 5), \bot$	equiv_pos2
8.	▷	$\bot$	(resolution 1, 5, 6, 7)

Contexts. A specificity of the Alethe proofs is the step context. Alethe contexts are a general mechanism to write substitutions and to change them by attaching new elements. A context is a possible empty list $c_1, \ldots, c_l$, where each element is either a variable or a variable-term tuple denoted $x_i \mapsto t_i$. In the first case, we say that c_i *fixes* the variable. The second case is a mapping. Throughout this chapter, Γ denotes an arbitrary context.

Every context Γ induces a substitution $\text{subst}(\Gamma)$. If Γ is the empty list, $\text{subst}(\Gamma)$ is the empty substitution, i.e, the identity function. The first case fixes x_n and allows the context to shadow a previously defined substitution for x_n:

$$\text{subst}([c_1, \ldots, c_{n-1}, x_n]) \text{ is } \text{subst}([c_1, \ldots, c_{n-1}]), \text{ but } x_n \text{ maps to } x_n.$$

When Γ ends in a mapping, the substitution is extended with this mapping:

$$\text{subst}([c_1, \ldots, c_{n-1}, x_n \mapsto t_n]) = \text{subst}([c_1, \ldots, c_{n-1}]) \circ \{x_n \mapsto t_n\}.$$

The following example illustrates this idea:

$$\text{subst}([x \mapsto 7, x \mapsto g(x)]) = \{x \mapsto g(7)\}$$
$$\text{subst}([x \mapsto 7, x, x \mapsto g(x)]) = \{x \mapsto g(x)\}$$

Contexts are used to express proofs about the preprocessing of terms. The conclusions of proof rules that use contexts always have the form

i.	Γ	$\triangleright$	$t \approx u$	(rule, ...)

where the term t is the original term, and u is the term after preprocessing. Tautologies with contexts correspond to judgments $\models_T \text{subst}(\Gamma)(t) \approx u$.

Formally, the context can be translated to λ-abstractions and applications. This is discussed in the Section 3.1.1.2.

Example 16
This example shows a proof that uses a subproof with a context to rename a bound variable.

1.		$\triangleright$	$\forall x.\,(P\ x)$	assume
2.		$\triangleright$	$\neg(\forall y.\,(P\ y))$	assume
3.	$y, x \mapsto y$	$\triangleright$	$x \approx y$	refl
4.	$y, x \mapsto y$	$\triangleright$	$(P\ x) \approx (P\ y)$	(cong 3)
5.		$\triangleright$	$\forall x.\,(P\ x) \approx \forall y.\,(P\ y)$	bind
6.		$\triangleright$	$\neg(\forall x.\,(P\ x) \approx \forall y.\,(P\ y))$, $\neg(\forall x.\,(P\ x)), (\forall y.\,(P\ y))$	equiv_pos2
7.		$\triangleright$	$\bot$	(resolution $1, 2, 5, 6$)

Implicit Reordering of Equalities. In addition to the explicit steps, solvers might reorder equalities, i.e., apply symmetry of the equality predicate, without generating steps. The sole exception is the topmost equality in the conclusion of steps with non-empty context. The order of the arguments of this equality can never change. As described above, all rules that accept a non-empty context have a conclusion of the form $t \approx u$. Since the context represents a substitution applied to the left-hand side, this equality symbol is not symmetric.

The SMT solver veriT currently applies this freedom in a restricted form: equalities are reordered only when the term below the equality changes during proof search. One such case is the instantiation of universally quantified variables. If an instantiated variable appears below an equality, then the equality might have an arbitrary order after instantiation. Nevertheless, consumers of Alethe must consider the possible implicit reordering of equalities everywhere.

3.1.1.1 The Syntax

The concrete text representation of the Alethe proofs is based on the SMT-LIB standard. Figure 3.1 shows an example proof as printed by veriT with light edits for readability. The format follows the SMT-LIB standard when possible. Input problems in the SMT-LIB formats are scripts. A SMT-LIB script is a list of commands that manipulate the SMT solver. For example, `assert` introduces an assertion, `check-sat` starts solving, and `get-proof` instructs the SMT solver to print the proof. Alethe mirrors this structure. Therefore, beside the SMT-LIB logic and term language, it also uses commands to structure the proof. An Alethe proof is a list of commands.

Usually, an Alethe proof is associated with a concrete SMT-LIB problem that is proved by the Alethe proof. This can either be a concrete problem file or, if the incremental solving commands of SMT-LIB are used, the implicit problem constructed at the invocation of a `get-proof` command. In this document, we will call this SMT-LIB problem the *input problem*. All symbols declared or defined in the input problem remain declared or defined, respectively. Furthermore, the symbolic names introduced by the `:named` annotation also stay valid and can be used in the script. For the purpose of the proof rules, terms are treated as if proxy names introduced by `:named` annotations have been expanded and annotations have been removed. For

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 vr4)))
(step t9.t1 (cl (= z2 vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p vr4)))
        :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((vr4 U)) (p vr4))))
        :rule bind)
...
(step t14 (cl (forall ((vr5 U)) (p vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((vr5 U)) (p vr5)))
                  (p a)))
          :rule forall_inst :args ((:= vr5 a)))
(step t16 (cl (not (forall ((vr5 U)) (p vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Figure 3.1 Example proof output. Assumptions are introduced; a subproof renames bound variables; the proof finishes with instantiation and resolution steps.

example, the term `(or (! (P a) :named baz) (! baz :foo))` and `(or (P a) (P a))` are considered to be syntactically equal. Here `:foo` is an arbitrary SMT-LIB annotation.

Figure 3.2 shows the grammar of the proof text. It is based on the SMT-LIB grammar, as defined in the SMT-LIB standard [17 Appendix B]. The non-terminals ⟨attribute⟩, ⟨function_def⟩, ⟨sorted_var⟩, ⟨symbol⟩, and ⟨term⟩ are as defined in the standard. The non-terminal ⟨proof_term⟩ corresponds to the ⟨term⟩ non-terminal of SMT-LIB, but is extended with the additional production for the `choice` binder.

Alethe proofs are a list of commands. The `assume` command introduces a new assumption. While this command could also be expressed using the `step` command with a special rule, the special semantic of an assumption justifies the presence of a dedicated command: assumptions are neither

```
⟨proof⟩ := ⟨proof_command⟩*
⟨proof_command⟩ := (assume ⟨symbol⟩ ⟨proof_term⟩ )
                 | (step ⟨symbol⟩ ⟨clause⟩ :rule ⟨symbol⟩
                     ⟨premises_annotation⟩?
                     ⟨context_annotation⟩? ⟨attribute⟩* )
                 | (anchor :step ⟨symbol⟩
                     ⟨args_annotation⟩? ⟨attribute⟩* )
                 | (define-fun ⟨function_def⟩ )
⟨clause⟩ := (cl ⟨proof_term⟩* )
⟨proof_term⟩ := ⟨term⟩ extended with
                 (choice ( ⟨sorted_var⟩ ) ⟨proof_term⟩ )
⟨premises_annotation⟩ := :premises ( ⟨symbol⟩+)
⟨args_annotation⟩ := :args ( ⟨step_arg⟩+ )
⟨step_arg⟩ := ⟨symbol⟩ | ( ⟨symbol⟩ ⟨proof_term⟩ )
⟨context_annotation⟩ := :args ( ⟨context_assignment⟩+ )
⟨context_assignment⟩ := ( ⟨sorted_var⟩ )
                      | (:= ⟨symbol⟩ ⟨proof_term⟩ )
```

Figure 3.2 The proof grammar.

tautological nor derived from premises. The `step` command, on the other hand, introduces a derived or tautological clause. Both commands `assume` and `step` require an index as the first argument to later refer back to it. This index is an arbitrary SMT-LIB symbol. It must be unique for each `assume` and `step` command. A special restriction applies to the `assume` commands not within a subproof, those reference assertions in the input SMT-LIB problem. To simplify proof checking, the `assume` command must use the name assigned to the asserted formula if there is any. For example, if the input problem contains `(assert (! (P c) :named foo))`, then the proof must refer to this

assertion (if it is needed in the proof) as (`assume foo (P c)`). Note that a SMT-LIB problem can assign a name to a term at any point, not only at its first occurrence. If a term has more than one name, any can be picked.

The second argument of `step` and `assume` is the conclusion of the command. For a `step`, this term is always a clause. To express disjunctions in SMT-LIB the `or` operator is used. This operator, however, needs at least two arguments and cannot represent unary or empty clauses. To circumvent this, we introduce a new `cl` operator. It corresponds to the standard `or` function extended to one argument, where it is equal to the identity, and zero arguments, where it is equal to `false`. The `anchor` and `define-fun` commands are used for subproofs and sharing, respectively. The `define-fun` command corresponds exactly to the `define-fun` command of the SMT-LIB language.

Furthermore, the syntax uses annotations as used by SMT-LIB. The original SMT-LIB syntax uses the non-terminal ⟨attribute⟩. The Alethe syntax uses some predefined annotation. To simplify parsing, the order in which those must be printed is strict. The `:premises` annotation denotes the premises and is skipped if the rule does not require premises. If the rule carries arguments, the `:args` annotation is used to denote them. Anchors have two annotations: `:step` provides the name of the step that concludes the subproof and `:args` provides the context as sorted variables and assignments. Note that in this annotation, the ⟨symbol⟩ non-terminal is intended to be a variable. After those pre-defined annotations, the solver can use additional annotations. This can be used for debugging, or other purposes. A consumer of Alethe proofs *must* be able to parse proofs that contain such annotations.

Subproofs The abstract notation denotes subproofs by marking them with a vertical line. To map this notation to a list of commands, Alethe uses the `anchor` command. This command indicates the start of a subproof. A subproof is concluded by a matching `step` command. This step must use a *concluding rule* (such as subproof, bind, and so forth).

After the `anchor` command, the proof uses `assume` commands to list the assumptions of the subproof. Subsequently, the subproof is a list of `step` commands that can use prior steps in the subproofs as premises.

In the abstract notation, every step is associated with a context. The concrete syntax uses anchors to optimize this. The context is manipulated in a nested way: if a step pops $c_1, \ldots, c_n$ from a context Γ, there is an earlier step which pushes precisely $c_1, \ldots, c_n$ onto the context. Since contexts can only be manipulated by push and pop, context manipulations are nested.

The anchor commands push onto the context and the corresponding step commands pop from the context. To indicate these changes to the context, every anchor is associated with a list of fixed variables and mappings. This list can be empty.

The :step annotation of the anchor command is used to indicate the step command that will end the subproof. The clause of this step command is the conclusion of the subproof. While it is possible to infer the step that concludes a subproof from the structure of the proof, the explicit annotation simplifies reconstruction and makes the proof easier to read. If the subproof uses a context, the :args annotation of the anchor command indicates the arguments added to the context for this subproof. The annotation provides the sort of fixed variables.

In the example proof (Figure 3.1) a subproof starts at the anchor command. It ends with the bind steps that finishes the proof for the renaming of the bound variable z2 to vr4.

A further restriction applies: only the conclusion of a subproof can be used as a premise outside the subproof. Hence, a proof checking tool can remove the steps of the subproof from memory after checking it.

Example 17
This example shows the proof from Example 16 expressed as a concrete proof.

```
(assume h1 (forall ((x S)) (P x)))
(assume h2 (not (forall ((y S)) (P y))))
(anchor :step t5 :args ((:= x y)))
(step t3 (cl (= x y)) :rule refl)
(step t4 (cl (= (P x) (P y))) :rule cong :premises (t3))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
                                :rule bind)
(step t6 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y)))
             (not (forall ((x S)) (P x)))
             (forall ((y S)) (P y)))) :rule equiv_pos2)
(step t7 (cl) :rule resolution :premises (h1 h2 t5 t6))
```

Alethe Proof Printing States Figure 3.3 shows the states of an Alethe proof abstractly. To generate a proof, the SMT solver must be in the *Unsat mode*, i.e., the solver determined that the problem is unsatisfiable. The SMT-LIB problem script then requests the proof by invoking the get-proof command.

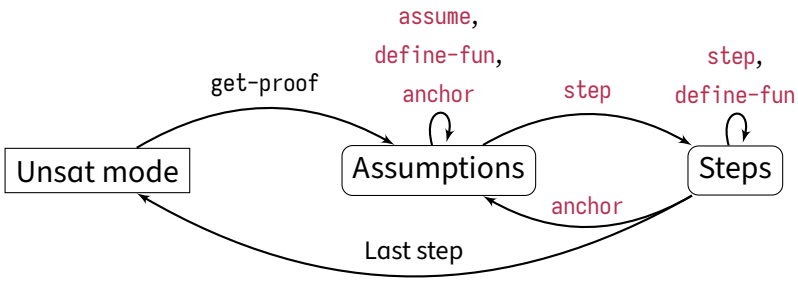


Figure 3.3 Abstract view of the transitions in an Alethe proof.

It is possible that this command fails. For example, if proof production was not activated up front. If there is no error, the proof is printed and printing starts with the assumptions. The solver prints the proof as a list of commands according to the state. The states ensure one constraint is maintained: assumptions can only appear at either the beginning of a proof or right after a subproof is started by the `anchor` command. They cannot be mixed with ordinary proof steps. This simplifies reconstruction. Each assumption printed at the beginning of the proof corresponds to assertions in the input problem, up to symmetry of equality. Proof printing concludes after the last step is printed and the solver returns to the Unsat mode and the user can issue further commands. Usually the last step is an outermost step (i.e., not within a subproof) that concludes the proof by deriving the empty clause, but this is not necessary. The solver is allowed to print some additional, useless, steps after the proof is concluded.

Sharing and Skolem Terms Usually, SMT solvers store terms internally in an efficient manner. A term data structure with perfect sharing ensures that every term is stored in memory precisely once. When printing the proof, this compact storage is unfolded. This leads to a blowup of the proof size.

Alethe can optionally use sharing[10] to print common subterms only once. This is realized using the standard naming mechanism of SMT-LIB. A term t is annotated with a name n by writing `(! t :named n )` where n is a symbol. After a term is annotated with a name, the name can be used in place of the term. This is a purely syntactical replacement. Alethe continues to use the names already used in the input problem. Hence, terms that already have a name in the input problem can be replaced by that name and new

[10] For veriT this can be activated by the command-line option `--proof-with-sharing`.

names introduced in the proof must not use names already used in the input problem.

To limit the number of names, an SMT solver can use the following simple approach used by veriT. Before printing the proof, it iterates over all terms of the proof and recursively descend into the terms. It marks every unmarked subterm it visits. If a visited term is already marked, the solver assigns a new name to this term. If a term already has a name, it does not descend further into the term. By doing so, it ensures that only terms that appear as child of two different parent terms get a name. Since a named term is replaced with its name after it first appearance, a term that only appears as a child of one single term does not need a distinct name. Thanks to the perfect sharing representation, testing if a term is marked takes constant time and the overall traversal takes linear time in the proof size.

To simplify reconstruction, Alethe can optionally[11] define Skolem constants as functions. In this case, the proof contains a list of `define-fun` commands that define shorthand 0-ary functions for the (`choice` ...) terms needed. Without this option, no `define-fun` commands are issued, and the constants are expanded.

Implicit Transformations Overall, the following aspects are treated implicitly by Alethe.

- Symmetry of equalities that are not top-most equalities in steps with non-empty context.
- The order of literals in the clauses.
- The unfolding of names introduced by (`!` *t* `:named` *s*).
- The removal of other SMT-LIB annotations of the form (`!` *t* ...).
- The unfolding of function symbols introduced by `define-fun`.[12]

Alethe proofs contain steps for other aspects that are commonly left implicit, such as renaming of bound variables, and the application of substitutions.

[11] For veriT by using the command-line option `--proof-define-skolems`.

[12] For veriT this is only used when the user introduces veriT to print Skolem terms as defined functions. User defined functions in the input problem are not supported by veriT in proof production mode.

3.1.1.2 Checking Alethe Proofs

In this section we present an abstract procedure to check if an Alethe proof is well-formed and valid. An Alethe proof is well-formed only if its anchors and steps are balanced. To check that this is the case, we replace innermost subproofs by holes until there is no subproof left. If the resulting proof is free of anchors and steps that use concluding rules, then the proof is well-formed. If all the steps in the subproofs adhere to the conditions of their rules, the subproof is valid. If all subproofs are valid, then the entire proof is valid.

Formally, an Alethe proof P is a list $[C_1, \ldots, C_n]$ of steps and anchors. Since every step uses an unique index, we assume that each step C_i in P uses i as its index. The context only changes at anchors and subproof-concluding steps. Therefore, the elements of $C_1, \ldots, C_n$ that are steps are not associated with a context. Instead, the context can be computed from the prior anchors. The anchors only ever extends the context.

To check an Alethe proof we can iteratively eliminate the first-innermost subproof, i.e., the innermost subproof that does not come after a complete subproof. The restriction to the first subproofs simplifies the calculation of the context of the steps in the subproof.

Definition 15 (First-Innermost Subproof)
Let P be the proof $[C_1, \ldots, C_n]$ and $1 \leq start < end \leq n$ be two indices such that

- C_{start} is an anchor,
- C_{end} is a step that uses a concluding rule,
- no C_k with $k < start$ uses a concluding rule,
- no C_k with $start < k < end$ is an anchor or a step that uses a concluding rule.

Then $[C_{start}, \ldots, C_{end}]$ is the first-innermost subproof of P.

Example 18
The proof in Example 17 has only one subproof and this subproof is also a first-innermost subproof. It is the subproof

```
(anchor :step t5 :args ((:= x y)))
(step t3 (cl (= x y)) :rule refl)
```

```
(step t4 (cl (= (P x) (P y))) :rule cong :premises (t3))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
                                :rule bind)
```

It is easy to calculate the context of the first-innermost subproof.

Definition 16 (Calculated Context)
Let $[C_{start}, \ldots, C_{end}]$ be the first-innermost subproof of P. For $start \leq i < end$, let $A_1, \ldots, A_m$ be the anchors among $C_1, \ldots, C_{i-1}$.
The calculated context of C_i is the context

$$c_{1,1}, \ldots, c_{1,n_1}, \ldots, c_{m,1}, \ldots, c_{m,n_m}$$

where $c_{k,1}, \ldots, c_{k,n_k}$ is the list of fixed variables and mappings associated with A_k.

Note that if C_i is an anchor, its calculated context does not contain the elements associated with C_i. Therefore, the context of C_{start} is the context of the steps before the subproof. Furthermore, the step C_{end} is the concluding step of the subproof and must have the same context as the steps surrounding the subproof. Hence, the context of C_{end} is the calculated context of C_{start}.

Example 19
The calculated context of the steps `t3` and `t5` in Example 17 is the context $x \mapsto y$. The calculated context of the concluding step `t5` and the anchor is empty.

A first-innermost subproof is valid if all its steps adhere to the conditions of their rule and only use premises that occur in front of them in the subproof. The conditions of each rule are listed the Appendix “The Alethe Proof Rules”.

Definition 17 (Valid First-Innermost Subproof)
Let $[C_{start}, \ldots, C_{end}]$ be the first-innermost subproof of P. The subproof is *valid* if

- all steps C_i with $start < i < end$ only use premises C_j with $start < j < i$,
- all C_i that are steps adhere to the conditions of their rule under the calculated context of C_i,
- the step C_{end} adheres to the conditions of its rule under the calculated context of C_{start}.

Since the assume rule expects an empty context, an admissible subproof can contain assumptions only if the rule used by C_{end} is the subproof rule.

To eliminate a subproof we can replace the subproof with a hole that has at its conclusion the conclusion of the subproof. This is safe as long as the subproof that is eliminated is valid (see Section 3.1.1.4).

Definition 18
The function E eliminates the first-innermost subproof from a proof if there is one. Let P be a proof $[C_1, \ldots C_n]$. Then $E(P) = P$ if P has no first-innermost subproof. Otherwise, P has the first-innermost subproof $[C_{start}, \ldots, C_{end}]$, and $E(P) = [C_1, \ldots, C_{start-1}, C', C_{end+1}, \ldots, C_n]$ where C' is a new step that uses the hole rule and has the index, conclusion, and premises of C_{end}.

It is important to add the premises of C_{end} to C'. The let rule can use additional premises and omitting those premises results in an unsound step. We can apply E iteratively to a proof P until we reach the least fixed point. Since $P$is finite we will always reach a fixed point in finitely many steps. The result is a list $[P_0, P_1, P_2, \ldots, P_{last}]$ where $P_0 = P$, $P_1 = E(P)$, $P_2 = E(E(P))$ and $P_{last} = E(P_{last})$.

Example 20
Applying E to the proof in Example 17 gives us the proof

```
(assume h1 (forall ((x S)) (P x)))
(assume h2 (not (forall ((y S)) (P y))))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
                                        :rule hole)
(step t6 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y)))
             (not (forall ((x S)) (P x)))
             (forall ((y S)) (P y)))) :rule equiv_pos2)
(step t7 (cl) :rule resolution :premises (h1 h2 t5 t6))
```

Since this proof contains no subproof, it is also P_{last}.

Definition 19 (Well-Formed Proof)
The Alethe proof P is well-formed if every step uses a unique index and P_{last} contains no anchor or step that uses a concluding rule.

Definition 20 (Valid Alethe Proof)
The proof P is a *valid Alethe proof* if

- P is well-formed,
- P does not contain any step that uses the hole rule,
- P_{last} contains a step that has the empty clause as its conclusion,
- the first-innermost subproof of every P_i, $i < last$ is valid,
- all steps C_i in P_{last} only use premises C_j in P_{last} with $1 \leq j < i$,
- all steps C_i in P_{last} adhere to the conditions of their rule under the empty context.

The condition that P contains no hole ensures that the original proof is complete and holes are only introduced by eliminating valid subproofs.

Example 21
The proof in Example 17 is valid. The only subproof is valid, the proof contains no hole, and P_{last} contains the step t7 that concludes with the empty clause.

It is sometimes useful to speak about the steps that are not within a subproof. We call such a step an *outermost step*. In a well-formed proof those are the steps of P_{last}.

3.1.1.3 Contexts and Metaterms

We now direct our attention to subproofs with contexts. It is useful to give a precise semantic to contexts to have the tools to check that rules that use contexts are sound. Contexts are local in the sense that they affect only the proof step they are applied to. For the full details on contexts see [12]. The presentation here is adapted from this publication, but omits some details.

To handle subproofs with contexts, we translate the contexts into λ-terms. This allows us to leverage the λ-calculus as an existing well-understood theory of binders. These λ-terms are called *metaterms*.

Definition 21 (Metaterm)
Metaterms are expressions constructed by the grammar

$$M ::= \boxed{t} \mid \lambda x.\, M \mid (\lambda \bar{x}_n.\, M)\, \bar{t}_n$$

where t is an ordinary term and t_i and x_i have matching sorts for all $0 \leq i \leq 1$.

According to this definition, a metaterm is either a boxed term, a λ-abstraction, or an application to an uncurried λ-term. The annotation $\boxed{t}$ delimits terms from the context, a simple λ-abstraction is used to express fixed variables, and the application expresses simulations substitution of n terms.[13]

We use $=_{\alpha\beta}$ to denote syntactic equivalence modulo α-equivalence and β-reduction.

Proof steps with contexts can be encoded into proof steps with empty contexts, but with metaterms. A proof step

$$\text{i. } \Gamma \quad \triangleright \quad t \approx u \quad (\text{rule } \bar{p}_n)\ [\bar{a}_m]$$

is encoded into

$$\text{i. } \quad \triangleright \quad L(\Gamma)[t] \simeq R(\Gamma)[u] \quad (\text{rule } \bar{p}_n)\ [\bar{a}_m]$$

where

$$\begin{aligned} L(\varnothing)[t] &= \boxed{t} & R(\varnothing)[u] &= \boxed{u} \\ L(x, \bar{c}_m)[t] &= \lambda x.\, L(\bar{c}_m)[t] & R(x, \bar{c}_m)[u] &= \lambda x.\, R(\bar{c}_m)[u] \\ L(\bar{x}_n \mapsto \bar{s}_n, \bar{c}_m)[t] &= (\lambda \bar{x}_n.\, L(\bar{c}_m)[t])\bar{s}_n & R(\bar{x}_n \mapsto \bar{s}_n, \bar{c}_m)[u] &= R(\bar{c}_m)[u] \end{aligned}$$

To achieve the same effect as using the subst() function described above, we can translate the terms into metaterms, perform β-reduction, and rename bound variables if necessary [12 Lemma 11].

Example 22
The example on page 44 becomes

$$L(x \mapsto 7, x \mapsto g(x))[x] = (\lambda x.\, (\lambda x.\, \boxed{x})\, (g(x)))\, 7 =_{\alpha\beta} \boxed{g(7)}$$
$$L(x \mapsto 7, x, x \mapsto g(x))[x] = (\lambda x.\, \lambda x.\, (\lambda x.\, \boxed{x})\, (g(x)))\, 7 =_{\alpha\beta} \lambda x.\, \boxed{g(x)}$$

Most proof rules that operate with contexts can easily be translated into proof rules using metaterms. The exception are the tautologous rules, such as refl and the ⋯_simplify rules.

[13] The box annotation used here is unrelated to the boxes within the SMT solver discussed in the introduction.

Steps that use such rules always encode a judgment $\vDash \Gamma \triangleright t \approx u$. With the encoding described above we get $L(\Gamma)[t] \approx R(\Gamma)[u] =_{\alpha\beta} \lambda \bar{x}_n.\, \boxed{t'} \approx \lambda \bar{x}_n.\, \boxed{u'}$ with some terms t', u'. To handle those terms, we use the reify() function. This function is defined as

$$\text{reify}(\lambda \bar{x}_n.\, \boxed{t} \approx \lambda \bar{x}_n.\, \boxed{u}) = \forall \bar{x}_n.\, (t \approx u).$$

Therefore, all tautological rules with contexts represent a judgment $\vDash \text{reify}(T \approx U)$ where $T =_{\alpha\beta} L(\Gamma)[t]$ and $U =_{\alpha\beta} R(\Gamma)[u]$.

Example 23
Consider the step

$$\text{i.}\quad y, x \mapsto y \quad \triangleright \quad x + 0 \approx y \qquad \text{sum\_simplify}$$

Translating the context into metaterms leads to

$$\text{i.}\quad \triangleright \quad (\lambda y.\, (\lambda x.\, \boxed{x + 0})\, y) \approx (\lambda y.\, \boxed{y}) \qquad \text{sum\_simplify}$$

Applying β-reduction leads to

$$\text{i.}\quad \triangleright \quad (\lambda y.\, \boxed{y + 0}) \approx (\lambda y.\, \boxed{y}) \qquad \text{sum\_simplify}$$

Finally, using reify() leads to

$$\text{i.}\quad \triangleright \quad \forall y.\, (y + 0 \approx y) \qquad \text{sum\_simplify}$$

This obviously holds in the theory of linear arithmetic.

3.1.1.4 Soundness

Any proof calculus should be sound. In the case of Alethe, most proof rules are standard rules, or simple tautologies. The rules that use context are unusual, but those proof rules were previously shown to be sound [12]. Alethe does not use any rules that are merely satisfiability preserving. The skolemization rules replace the bound variables with choice terms instead of fresh symbols.[14] All Alethe rules express semantic implications. Overall, we assume in this document that the proof rules and proofs written in the abstract notation are sound.

[14] The `define-fun` function can introduce fresh symbols, but we will assume here that those commands have been eliminated by unfolding the definition.

Nevertheless, a modest gap remains. The concrete, command-based syntax does not precisely correspond to the abstract notation. In this section we will address the soundness of concrete Alethe proofs.

Theorem 1 (Soundness of Concrete Alethe Proofs)
If there is a valid Alethe proof $P = [C_1, \ldots, C_n]$ that has the formulas $\varphi_1, \ldots, \varphi_m$ as the conclusions of the outermost assume steps, then

$$\varphi_1, \ldots, \varphi_m \models \bot.$$

Here, $\models$ represents semantic consequence in the many-sorted first order logic of SMT-LIB with the theories of uninterpreted functions and linear arithmetic extended to clauses.

To show the soundness of a valid Alethe proof $P = [C_1, \ldots, C_n]$, we can use the same approach as for the definition of validity: consider first-innermost subproof first and then replace them by holes. Since valid proofs do not contain holes, we have to generalize the induction to allow holes that were introduced by the elimination of subproofs. We start with simple subproofs with empty contexts and without nested subproofs.

Lemma 1
Let P be a proof that contains a valid first-innermost subproof where C_{end} is a subproof step. Let ψ be the conclusion of C_{end}. Then $\models \psi$ holds.

Proof. First, we use induction on the number of steps n after C_{start}. Let ψ_n be the conclusion of $C_{start+n}$ and V_n the conclusions of the assume steps in $[C_{start}, \ldots, C_{start+n}]$. Assumptions always introduce unit clauses. We will identify unit clauses with their single literal. We will show $V_n \models \psi_n$ if $start + n < end$.

If $n = 1$, then $C_{start+n} = C_{start+1}$ must either be a tautology, or an assume step. In the first case, $\models \psi_{start+1}$ holds, and in the second case $\psi_{start+1} \models \psi_{start+1}$ holds.

For subsequent n, $C_{start+n}$ is either an ordinary step, or an assume step. In the second case, $\psi_{start+n} \models \psi_{start+n}$ which can be weakened to $V_n \models \psi_{start+n}$. In the first case, the step $C_{start+n}$ has a set of premises S. For each step $C_{start+i} \in S$ we have $i < n$ and $V_i \models \psi_{start+i}$ due to the induction hypothesis. Since $i < n$, the premises V_i are a subset of V_n and we can weaken $V_i \models \psi_{start+i}$ to $V_n \models \psi_{start+i}$. Since all premises of $C_{start+n}$ are the consequence of V_n we get $V_n \models \psi_n$.

The step C_{end-1} is the last step of the subproof that does not use a concluding rule. At this step we have $V_{end-1} \models \psi_{end-1}$. Since C_{end} is not an assume step, the set $V_{end-1} = \{\varphi_1, \ldots, \varphi_m\}$ contains all assumptions in the subproof. By the deduction theorem we get

$$\models \varphi_1 \wedge \cdots \wedge \varphi_m \rightarrow \psi_{end-1}.$$

This can be transformed into the clause

$$\models \neg\varphi_1, \cdots, \neg\varphi_m, l_1, \ldots, l_o.$$

where $l_1, \ldots, l_o$ are the literals of ψ_{end-1}. This clause is exactly the conclusion of the step C_{end} according to the definition of the subproof rule. □

We can do the same reasoning as for Lemma 1 for subproofs with contexts. This is slightly complicated by the let rule that can use extra premises.

Lemma 2
Let P be a proof that contains a valid first-innermost subproof where C_{end} is a step using one of: bind, sko_ex, sko_forall, onepoint, let.
Then $V \models \Gamma \triangleright \psi$ where Γ is the calculated context of C_{start} and ψ is the conclusion of C_{end}. The set V is empty if C_{end} does not use the let rule. Otherwise, it contains all conclusions of the assume steps among $[C_\delta, \ldots, C_{start}]$ where δ is either the largest index $\delta < start$ such that s_δ is an anchor, or 1 if no such index exist. Hence, there is no anchor between C_δ and C_{start}.

Proof. The step C_{start} is an anchor due to the definition of innermost-first subproof. Let $c_1, \ldots, c_n$ be the context introduced by the anchor C_{start}, and let Γ be the calculated context of C_{start}. $\Gamma' = \Gamma, c_1, \ldots, c_n.$ is the calculated context of the steps in the subproof after C_{start}.

The step C_{end} is a step

$$\begin{array}{rl|lll} & & & \cdots & \\ end-1. & \Gamma' \quad \triangleright & & \psi' & (\ldots) \\ \hline end. & \Gamma \quad \triangleright & & \psi & (\text{rule } i_1, \ldots, i_n) \end{array}$$

Since we assume the step C_{end} is correctly employed, $\models \Gamma \triangleright \psi$ holds, as long as $\models \Gamma' \triangleright \psi'$ holds.

We perform the same induction as for Lemma 1 over the steps in $[C_{start}, \ldots, C_{end}]$. Since C_{end} does not use the subproof rule, the subproof

does not contain any assumptions and V_i stays empty. Again, we are interested in the step C_{end-1}. At this step we get $\vDash \Gamma' \rhd \psi'$.

Only the let rule uses additional premises $C_{i_1}, \ldots, C_{i_n}$. Hence, for all other rules, the conclusion cannot depend on any step outside the subproof and V is empty. Due to the definition of first-innermost subproof, all steps $C_{i_1}, \ldots, C_{i_n}$ are in the same subproof that starts at C_δ.

The steps $C_{i_1}, \ldots, C_{i_n}$ might depend on some assume steps that appear before them in their subproof. This is the case if the steps are outermost steps, or if the subproof that starts at C_δ concludes with a subproof step. In this case we can, as we saw in the proof of Lemma 1, weaken their judgments to include all assumptions in $[C_\delta, \ldots, C_{start}]$.

If the subproof that starts at C_δ concludes with any other rule, then there cannot be any assumptions and V is empty. □

By using Lemma 1 and Lemma 2 we can now show that a valid, concrete Alethe proof is sound. That is, we can show Theorem 1.

Proof. Since $P = [C_1, \ldots, C_n]$ is valid, all steps that do not use the hole rule adhere to their rule. Since we assume that the abstract notation and the rules are sound, we only have to worry about the steps using the hole rule. Those should be sound, i.e., for a hole step with the conclusion ψ, premises V, and context Γ the judgment $V \vDash \Gamma \rhd \psi$ must hold.

Since P is a valid proof there is a sequence $[P_0, \ldots, P_{last}]$ as discussed in Section 3.1.1.2. For $i < last$, $E(P_i) = P_{i+1}$ replaces the first-innermost subproof in P_i by a hole with the conclusion ψ. Furthermore, the context of the introduced hole corresponds to the context Γ of the start of the subproof. Since P is a valid proof, the first-innermost subproof eliminated by E is always valid. Therefore, we can apply Lemma 1 or Lemma 2 to conclude that the hole introduced by E is sound.

Since P_0 does not contain any holes, the holes in each proof P_i are all introduced by innermost-first subproof elimination. Therefore, they are sound. In consequence, all holes in P_{last} are sound and we can perform the same argument as in the proof of Lemma 1 to the proof P_{last}.

Let j be the index of the step in P_{last} that concludes with the empty clause. Let $start = 1$ and $end = j$ in the argument of Lemma 1. This shows that $V \vDash \bot$, where V is the conclusion of the assume steps in the sublist $[C_1, \ldots, C_j]$ of P_{last}. We can weaken this by adding the conclusions of the assume steps in $[C_j, \ldots, C_n]$ of P_{last} to get $\varphi_1, \ldots, \varphi_m \vDash \bot$. □

3.1.2 The Alethe Rules

Together with the language, the Alethe format also includes a set of proof rule. The Appendix "The Alethe Proof Rules" gives a full list of all proof rules. Currently, the proof rules correspond to the rules that the solver veriT can emit. For the rest of this section, we will discuss some general concepts related to the rules.

Tautologous Rules and Simple Deduction. Most rules introduce tautologies. One example is the and_pos rule: $\neg(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n), \varphi_i$. Other rules derive their conclusion from a single premise. Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the implies rule eliminates an implication: From $\varphi_1 \to \varphi_2$, it deduces $\neg\varphi_1, \varphi_2$.

Resolution. CDCL(T)-based SMT solvers, and especially their SAT solvers, are fundamentally based on resolution of clauses. Hence, Alethe also has dedicated clauses and a resolution proof rule. However, since SMT solvers do not enforce a strict clausal normal form, ordinary disjunction is also used. Keeping clauses and disjunctions distinct simplifies rule checking. For example, in the case of resolution there is a clear distinction between unit clauses where the sole literal starts with a disjunction and non-unit clauses. The syntax for clauses uses the `cl` operator, while disjunctions use the standard SMT-LIB `or` operator. The or *rule* is responsible for converting disjunctions into clauses.

The Alethe proofs use a generalized propositional resolution rule with the name resolution or th_resolution. Both names denote the same rule. The difference only serves to distinguish if the rule was introduced by the SAT solver or by a theory solver. The resolution step is purely propositional; there is no notion of a unifier. The resolution rules also implicitly simplifies repeated negations at the head of literals.

The premises of a resolution step are clauses, and the conclusion is a clause that has been derived from the premises by some binary resolution steps.

Quantifier Instantiation. To express quantifier instantiation, the rule forall_inst is used. It produces a formula of the form $(\neg\forall \bar{x}_n.\ \varphi) \vee \varphi[\bar{t}_n]$, where

φ is a term containing the free variables $\bar{x}_n$, and for each i the ground term t_i is a new term with the same sort as x_i.[15]

The arguments of a forall_inst step is the list $(x_1, t_1), \ldots, (x_n, t_n)$. While this information can be recovered from the term, providing it explicitly helps reconstruction because the implicit reordering of equalities obscures which terms have been used as instances. Existential quantifiers are handled by skolemization.

Linear Arithmetic. Proofs for linear arithmetic use a number of straightforward rules, such as la_totality ($t_1 \leq t_2 \vee t_2 \leq t_1$)[16] and the main rule la_generic. The conclusion of an la_generic step is a tautology $\neg\varphi_1, \neg\varphi_2, \ldots, \neg\varphi_n$ where the φ_i are linear (in)equalities. Checking the validity of this clause amounts to checking the unsatisfiability of the system of linear equations $\varphi_1, \varphi_2, \ldots, \varphi_n$. The annotation of an la_generic step contains a coefficient for each (in)equality. The result of forming the linear combination of the literals with the coefficients is a trivial inequality between constants.

Example 24
The following example is the proof for the unsatisfiability of $(x + y < 1) \vee (3 < x)$, $x \approx 2$, and $0 \approx y$.

1.	$\triangleright$	$(3 < x) \vee (x + y < 1)$	assume
2.	$\triangleright$	$x \approx 2$	assume
3.	$\triangleright$	$0 \approx y$	assume
4.	$\triangleright$	$(3 < x), (x + y < 1)$	(or 1)
5.	$\triangleright$	$\neg(3 < x), \neg(x \approx 2)$	la_generic [1.0, 1.0]
6.	$\triangleright$	$\neg(3 < x)$	(resolution 2, 5)
7.	$\triangleright$	$x + y < 1$	(resolution 4, 6)
8.	$\triangleright$	$\neg(x + y < 1), \neg(x \approx 2) \vee \neg(0 \approx y)$	la_generic [1.0, −1.0, 1.0]
9.	$\triangleright$	$\bot$	(resolution 8, 7, 2, 3)

[15] For historical reasons, forall has a unit clause with a disjunction as its conclusion and not the clause $(\neg\forall\bar{x}_n.\, \varphi), \varphi[\bar{t}_n]$.
[16] This rule also has a unit clause with a disjunction as its conclusion.

Skolemization and Other Preprocessing Steps. One typical example for a rule with context is the sko_ex rule that is used to express skolemization of an existentially quantified variable. The conclusion of a step that uses this rules is an equality. The left-hand side is a formula starting with an existential quantifier over some variable x. In the formula on the right-hand side, the variable is replaced by the appropriate Skolem term. To provide a proof for the replacement, the sko_ex step uses one premise. The premise has a context that maps the existentially quantified variables to the appropriate Skolem term.

i.	$\Gamma, x \mapsto (\varepsilon x.\, \varphi)$	$\triangleright$	$\varphi \approx \psi$	$(\ldots)$
j.	Γ	$\triangleright$	$(\exists x.\, \varphi) \approx \psi$	(sko_ex)

Example 25
To illustrate how such a rule is applied, consider the following example taken from [12]. Here the term $\neg p(\varepsilon x.\neg p(x))$ is skolemized. The refl rule expresses a simple tautology on the equality (reflexivity in this case), cong is functional congruence, and sko_forall works like sko_ex, except that the choice term is $\varepsilon x.\neg\varphi$.

1.	$x \mapsto (\varepsilon x.\, \neg(p\, x))$	$\triangleright$	$x \approx \varepsilon x.\, \neg(p\, x)$	refl
2.	$x \mapsto (\varepsilon x.\, \neg(p\, x))$	$\triangleright$	$(p\, x) \approx p(\varepsilon x.\, \neg(p\, x))$	(cong 1)
3.		$\triangleright$	$(\forall x.\, (p\, x)) \approx (p\, (\varepsilon x.\, \neg(p\, x)))$	(sko_forall 2)
4.		$\triangleright$	$(\neg\forall x.\, (p\, x)) \approx \neg(p\, (\varepsilon x.\, \neg(p\, x)))$	(cong 3)

3.2 Reconstructing Alethe Proofs in Isabelle

Joint work with Mathias Fleury and Martin Desharnais. This section was published at CADE 2021 [88]. I designed and implemented all changes on the SMT solver side and contributed to the evaluation.

Proof assistants are used in verification and formal mathematics to provide trustworthy, machine-checkable formal proofs of theorems. Proof *automation* reduces the burden of finding proofs and allows proof assistant users to focus on the core of their arguments instead of technical details. A successful approach is implemented by "hammers," like Sledgehammer for Isabelle/HOL [22]. As discussed in the introduction, Sledgehammer uses a multi-stage process. First, it heuristically selects facts from the background and use an external automatic theorem prover, such as a satisfiability modulo theories (SMT) solver [18], to filter facts needed to discharge the goal. Then it uses the filtered facts to find a trusted proof during preplay.

Isabelle/HOL does not accept proofs that do not go through the assistant's inference kernel. Hence, Sledgehammer attempts to find the fastest internal tactic that can recreate the proof (*preplay*). This is often a call of the `smt` tactic, which runs an SMT solver, parses the proof, and reconstructs it through the kernel. This reconstruction allows the usage of external provers. The `smt` tactic was originally developed for the SMT solver Z3 [28,47]. See Figure I.1 for an overview of the Sledgehammer workflow.

The SMT solver CVC4 is one of the best solvers on Sledgehammer generated problems [24], but does not produce proofs for problems with quantifiers. Recently cvc5 was released that features a new proof module that can generate proofs for problems with quantifiers. Currently, there is an effort to extend the work presented here to proofs generated by cvc5 (see Section 3.3.3). When CVC4 is successful during the first fact-filtering step of Sledgehammer, Sledgehammer often suggests the `smt` tactic based on Z3 during preplay. However, since CVC4 uses more elaborate quantifier instantiation techniques, many problems provable for CVC4 are unprovable for Z3. Therefore, Sledgehammer regularly fails to find a trusted proof and the user has to write the proofs manually. As discussed in Section 2.2.2, veriT [31] supports these techniques and we extend the `smt` tactic to reconstruct its proofs. With the new reconstruction (Section 3.2.1), more `smt` calls are successful. Hence, less manual labor is required from users.

The runtime of the `smt` tactic depends on the runtime of the reconstruction and the solver. To simplify the reconstruction, we do not treat veriT as a black box anymore, but extend it to produce more detailed proofs that are easier to reconstruct. We use detailed rules for simplifications with a combination of propositional, arithmetic, and quantifier reasoning. Similarly, we

add additional information to avoid search, e.g., for linear arithmetic and for term normalization. Our reconstruction method uses the newly provided information, but it also has a *step skipping* mode that combines and removes some steps (Section 3.2.2).

The development of the reconstruction evolved over multiple phases. A very early prototype of the extension was used to validate the fine-grained proof format itself [12 Section 6.2, second paragraph]. At this point I became involved in the efforts that had the goal of developing the very early prototype into a reliable and fast reconstruction method. We also reported on work in progress and published some details of the reconstruction method and the rules [54].

We optimize the performance further by tuning the search performed by veriT. Multiple options influence the execution time of an SMT solver. To fine-tune veriT's search procedure, we select four different combinations of options, or *strategies*, by generating typical problems and selecting options with complementary performance on these problems. We use the schedgen toolbox (Chapter V) for this task. We extend Sledgehammer to compare these four selected strategies and suggest the fastest to the user. We then evaluate the reconstruction with Sledgehammer on a large benchmark set. Our new tactic halves the failure rate. We also study the time required to reconstruct each rule. Many simple rules occur often, showing the importance of step skipping (Section 3.2.3).

Finally, we discuss related work (Section 3.3.1). Compared to the published work in progress, the `smt` tactic is now thoroughly tested. We fixed all issues revealed during development and improved the performance of the reconstruction method. The work presented here is integrated into Isabelle/HOL version 2021; i.e., since this version Sledgehammer can also suggest veriT, without user interaction.

3.2.1 Overview of the veriT-Powered SMT Tactic

Isabelle/HOL is a generic proof assistant based on an intuitionistic logic framework, *Pure*, and is almost always only used parameterized with a logic. In this work we use only Isabelle/HOL, the parameterization of Isabelle/HOL with higher-order logic with rank-1 (top level) polymorphism. Isabelle/HOL adheres to the LCF (Logic for Computable Functions) [58] tradition that started in the 1970s. Its kernel supports only a small number of inferences. Tactics are programs that prove a goal by using only the kernel for inferences.

The LCF tradition also means that external tools, like SMT solvers, are not trusted.

Nevertheless, external tools are successfully used. They provide relevant facts or a detailed proof. The Sledgehammer tool implements the former and passes the filtered facts to trusted tactics during preplay. The `smt` tactic implements the latter approach. The provided proof is checked by Isabelle/HOL. We extended the `smt` tactic with support for veriT's proofs, and extended Sledgehammer to try the extended `smt` tactic during preplay.

The `smt` tactic translates the current goal to the SMT-LIB format [17], runs an SMT solver, parses the proof, and replays it through Isabelle/HOL's kernel. To choose the `smt` tactic the user applies `(smt (z3))` to use Z3 and `(smt (verit))` to use veriT. We will refer to them as z-`smt` and v-`smt`. The proof formats of Z3 and veriT are so different that separate reconstruction modules are needed. The v-`smt` tactic performs four steps:

1. It negates the proof goal to have a refutation proof and also encodes the goal into first-order logic. The encoding eliminates λ-abstractions. To do so, it replaces each λ-abstraction with a new function and creates "*app*" operators corresponding to first-order binary function application. Although a veriT version with some higher-order support exists [16], it does not generate proofs yet. Then veriT is called to find a proof.

2. It parses the proof found by veriT (if one is found) and encodes it as a directed acyclic graph with $\bot$ as the only conclusion. This graph may contain subproofs with local assumptions. The proof structure is slightly amended to simplify reconstruction.

3. It converts the SMT-LIB terms to typed Isabelle/HOL terms and also reverses the encoding used to convert higher-order into first-order terms.

4. It traverses the proof graph, checks that all input assertions match their Isabelle/HOL counterpart and then reconstructs the proof step by step using the kernel's primitives.

Figure 3.4 shows a schema of the pipeline used by the `smt` tactic when reconstruction proofs generated by veriT. The code used in the steps marked with * is shared with the reconstruction for Z3 proofs [28]. The code for the step marked with † only shares the reconstruction of resolution steps

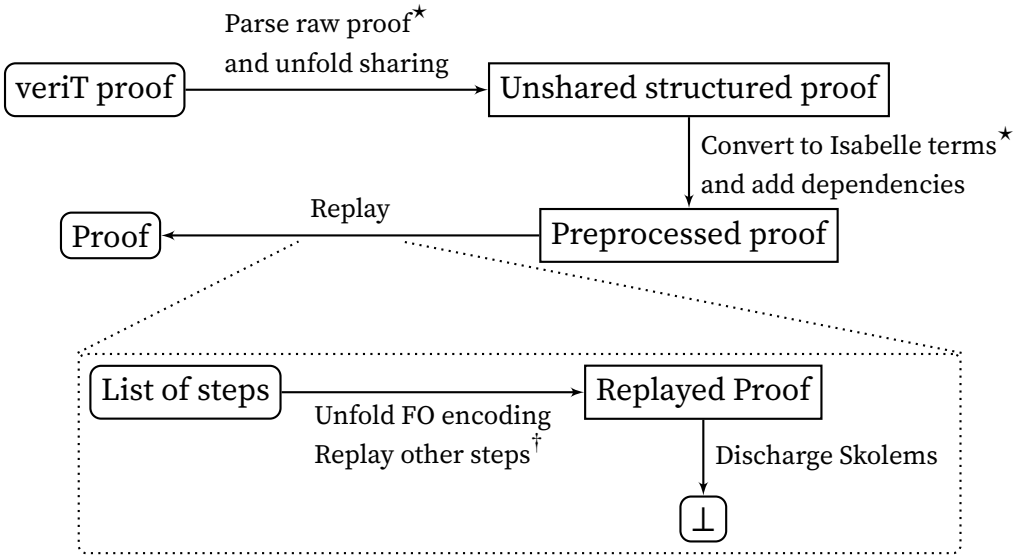


Figure 3.4 The reconstruction pipeline.

with the code for Z3 proofs. It was first introduced to show the efficacy of fine-grained preprocessing proofs [12], but changed substantially since then.

Isabelle/HOL's proof interaction is built around the *context*.[17] It encompasses all the symbols and definitions available globally and locally in the current proof. Assumptions (e.g., to introduce local assumption) and variables can extend contexts. Once a theorem is proven in an extended context, it can be exported back to the original context: new assumptions become as assumptions of the theorem and variables universally quantified.

3.2.1.1 Application of Theorems

Most rules that can be applied are either tautologies or applications of theorems that can be easily expressed: the rule true is the tautology used to prove that the theorem $\top$ holds. Similarly, the transitivity rule eq_transitive transforms the assumptions $(t_j \approx t_{j+1})_{j<n}$ into $t_0 \approx t_n$.

The reconstruction functions for tautologies and easy theorems share many characteristics.

[17] Isabelle/HOL contexts and Alethe contexts are unrelated concepts.

- The functions take additional information provided in the proof output, such as the substitution provided by the forall_inst rule, into account.

- The functions apply the theorem and then use `simp` to reorder the equalities and prove that the terms are equal. For the ite2 rule on $(\textbf{ite}\ \neg\varphi\ \psi_1\ \psi_2) \vee \varphi' \vee \psi_2'$, we identify the terms, φ, ψ_1, and ψ_2, and generate the tautology $(\textbf{ite}\ \neg\varphi\ \psi_1\ \psi_2) \vee \varphi \vee \psi_2$, that can be used by `simp` to discharge the goal by showing $\varphi \approx \varphi'$ and $\psi_2 \approx \psi_2'$. The search space is very large, and the search can be very time-consuming during the reconstruction.

In practice, there is a challenge: equalities can be reoriented. veriT applies the symmetry of equality transparently to ensure internal invariants are maintained. The reordering happens mostly when producing new terms (during parsing or instantiation). However, we do not want to rely on this specific behavior which could change in a future version.

3.2.1.2 Subproofs

Unlike Z3, veriT has subproofs. Subproofs fall into two categories: proofs used to justify proof steps (e.g., for skolemization) and lemmas with assumptions and fixed variables. In Isabelle/HOL, both are modelled by the notions of contexts that encapsulate all the assumptions and fixed variables present at a given point.

The first kind of subproofs are proofs of *lemmas* that come with additional assumptions. They are used for example for proofs like $P \to \bot$. The formula P is an assumption of the proof (given by an assume command) and $\bot$ is the conclusion.

In Isabelle/HOL, we start by extracting all the assumptions when entering the proof. This creates a new context. Then, we replay the proof in the new context. The assume commands are now entailed by the context are replayed as such. Finally, the conclusion is exported to the outer context.

Replaying *subproofs* is similar to replaying lemmas in the proof: we enter contexts with new assumptions and variables, depending on the rule. At the end of the subproof the last step is exported back to the outer context and is used to discharge the conclusion. For example, the subproof of a simple bind step will be of the form $(\forall x\ y.\ (x = y \to P\ x = Q\ y))$ to prove that $(\forall x.\ P\ x) = (\forall y.\ Q\ y)$.

3.2.2 Tuning the Reconstruction

To improve the speed of the reconstruction method, we create small and well-defined rules for preprocessing simplifications (Subsection 3.2.2.1). Previously, veriT implicitly normalized every step; e.g., repeated literals were immediately deleted. It now produces proofs for this transformation (Subsection 3.2.2.2). Finally, the linear-arithmetic steps contain coefficients which allow Isabelle/HOL to reconstruct the step without relying on its limited arithmetic automation (Subsection 3.2.2.3). On the Isabelle/HOL side, the reconstruction module selectively decodes the first-order encoding (Subsection 3.2.2.5). To improve the performance of the reconstruction, it skips some steps (Subsection 3.2.2.6).

3.2.2.1 Preprocessing Rules

During preprocessing SMT solvers perform simplifications on the operator level which are often akin to simple calculations; e.g., $a \times 0 \times (f\ x)$ is replaced by 0.

To capture such simplifications, we create a list of 17 new rules: one rule per arithmetic operator, one to replace Boolean operators such as **xor** with their definition, and one to replace n-ary operator applications with binary applications. This is a compromise: having one rule for every possible simplification would create a longer proof. Since preprocessing uses structural recursion, the implementation simply picks the right rule in each leaf case. The example above now produces a prod_simplify step with the conclusion $a \times 0 \times (f\ x) \approx 0$.

Previously, the single connect_equiv rule collected all those simplifications and no list of simplifications performed by this rule existed. The reconstruction relied on an experimentally created list of tactics to be fast enough.

On the Isabelle/HOL side, the reconstruction is fast, because we can direct the search instead of trying automated tactics that can also work on other parts of the formula. For example, the simplifier handles the numeral manipulations of the prod_simplify rule and we restrict it to only use arithmetic lemmas.

Moreover, since we know the performed transformations, we can ignore some parts of the terms by *generalizing*, i.e., replacing them by constants [28]. Because generalized terms are smaller, the search is more directed and we

are less likely to hit the search-depth limitation of Isabelle/HOL's `auto` tactic as before. Overall, the reconstruction is more robust and easier to debug.

3.2.2.2 Implicit Steps

To simplify the reconstruction, we avoid any implicit normal form of conclusions. For example, a rule concluding $\varphi \vee P$ for any formula φ and a predicate P can be used to prove $P \vee P$. In such cases, veriT automatically normalizes the conclusion $P \vee P$ to P. Before our work, veriT could not generate proofs for these normalization. Without a proof of the normalization, the reconstruction has to handle such cases. We add new proof rules for the normalization and extend veriT to use them. Since the conclusion of the new rules are not normalized, the code creating crating these steps has to bypass the normalization process.

Instead of keeping only the normalized step, both the original and the normalized step appear in the proof. For the example above, we have the step $P \vee P$ and the normalized P. To remove a double negation $\neg\neg\varphi$ we introduce the tautology $\neg\neg\neg\varphi \vee \varphi$ and resolve it with the original clause.

On the Isabelle/HOL side, the reconstruction becomes more regular with fewer special cases and is more reliable. The reconstruction method can directly reconstruct rules. Generalizing over the literals is possible, but is slow for small terms. To deal with the normalization, the reconstruction used to first generate the conclusion of the theorem and then ran the simplifier to match the normalized conclusion. This could not deal with tautologies. The extra step improves the reliability and the speed of the reconstruction.

Transformations can also lead to tautological clauses. For example, a step can translate $P \to P$ into $\neg P \vee P$, which was normalized to $\top$ without a proof. While it is possible to handle them by considering that almost every step or assumption can also be $\top$, a separate proof step for this normalization is simpler. Optionally, the solver now also prunes steps concluding $\top$.

We also improve the proof reconstruction of quantifier instantiation steps. One of the instantiation schemes, *conflicting instances*, only works on clausified terms. We introduce an explicit quantified-clausification rule qnt_cnf issued before instantiating. While this rule is not detailed, knowing when clausification is needed improves reconstruction, because it avoids clausifying unconditionally. The clausification is also shared between instantiations of the same term.

Our approach for the normalization was very different in the work in progress versions of the reconstruction [54]. For duplicates, we relied on the simplifier to remove them. We had not identified the issue with simplifications to $\top$, because they do not happen very often. To efficiently reconstruct connect_equiv that combines several rules, our work in progress version first attempted to reconstruct the steps using propositional logic only and then used Isabelle/HOL's auto tactic, because it also supports arithmetic and first-order transformation to some degree. The new detailed rules avoid such strange constructs.

One implicit transformation remains without proof: veriT applies the symmetry of equality implicitly. This *reordering* of equalities enforces the global invariant that for equality $\varphi_1 \approx \varphi_2$ the term index of φ_1 is less or equal than the term index of φ_2. The term index is the position of the term in the global term array. To enforce this invariant the API used to create equalities silently reorders arguments. Since this API is also used during parsing, such reordering is applied even before initialization of the proof module. The initial reordering is the motivation for repeating the input assertions in the proof. The assume steps in the printed proof correspond to the assertions in the SMT-LIB problem generated by Isabelle/HOL up to reordering of the equalities. The simplifier can easily prove that the assertion and the assumption are equal. This simplifies the reconstruction of subsequent proof steps, because proof steps do not reorder equalities often, in practice.

3.2.2.3 Arithmetic Reasoning

Proof Production. We use a proof witness to handle linear arithmetic. When the propositional model is unsatisfiable in the theory of linear real arithmetic, the solver creates la_generic steps. The conclusion is a tautological clause of linear inequalities and equations and the justification of the step is a list of coefficients so that the linear combination is a trivially contradictory inequality after simplification (e.g., $0 \geq 1$). Farkas' lemma guarantees the existence of such coefficients for reals. Most SMT solvers, including veriT, use the simplex method [44] to handle linear arithmetic. It calculates the coefficients during normal operation.

The real arithmetic solver also strengthens inequalities on integer variables before adding them to the simplex method. For example, if x is an

integer, the inequality $(2 \times x) < 3$ becomes $x \leq 1$. The corresponding justification is the rational coefficient ½. The reconstruction must replay this strengthening.

Since veriT's simplex stores the coefficients without a sort, we print them as decimals (e.g., `1.0`) whenever the input problem uses a theory with real numbers and otherwise resort to numerals (e.g., `1`) and integer division. It is the task of the reconstruction to correctly apply the coefficients.

The complete linear arithmetic proof step $1 < x \vee 2x < 3$ looks like

```
(step t11 (cl (< 1 x) (< (* 2 x) 3))
    :rule la_generic :args (1 (div 1 2)))
```

In proofs generated by Z3, coefficients are also produced for arithmetic steps called farkas-lemma. Z3 ignores them during the reconstruction. However, although the coefficients are rationals, they don't contain any strengthening because a separate step logs that.

Reconstruction. The reconstruction of an la_generic step in Isabelle/HOL starts with the goal $\bigvee_i \neg c_i$ where each c_i is either an equality or an inequality. The reconstruction method first generalizes over the non-arithmetic parts by replacing the terms in the equations by constants. The generalization corresponds to veriT's arithmetic solver which treats such terms as simplex variables and therefore it does not limit the reconstruction. Generalization also stops the reconstruction from looking inside terms. We encountered an example where a term of the form $((\textbf{ite}\ \top\ 0\ 1) < 0)$ appeared in a lemma and was prematurely simplified to $\bot$. Then we transform the lemma into the equivalent formulation $c_1 \to \ldots \to c_n \to \bot$ and remove all negations (e.g., replacing $\neg a \leq b$ by $b > a$).

Next, the reconstruction method multiplies the equation by the corresponding coefficient. For example, for integers, the inequality $A < B$, and the coefficient $\frac{p}{q}$ (with $p > 0$ and $q > 0$), it strengthens the equation and multiplies by p to get

$$p \times (\textbf{div}\ A\ q) + p \times (\textbf{ite}\ (\textbf{mod}\ B\ q \approx 0)\ 1\ 0) \leq p \times (\textbf{div}\ B\ q).$$

The if-then-else term $(\textbf{ite}\ (\textbf{mod}\ B\ q \approx 0)\ 1\ 0)$ corresponds to the strengthening. If $\textbf{mod}\ B\ q \approx 0$, the result is an equation of the form $A' + 1 \leq B'$, i.e., $A' < B'$. No strengthening is required for the corresponding theorem over reals.

Finally, we can combine all the equations by summing them while being careful with the equalities that can appear. Internally, to combine two equations (that involve $\leq$, $<$, or $\approx$), we currently do not try to find which theorem is needed for that combination of (in)equalities, but simply try to unify all possible theorems and see which single theorem remains at the end. Only one can remain. We simplify the resulting (in)equality using Isabelle/HOL's simplifier to derive $\bot$.

Coefficients are typed in Isabelle/HOL as either real or integer. The sort used by veriT is depending on the SMT-LIB theory and not on the current equation. To handle this, we use two versions of the equations to multiply by the coefficients, one with integers and one with rounding from reals to integers.

When handling the assumptions of the equations to multiply with the coefficients, we experimented with aggressive simplification and discharging all assumptions together. The former allows us to shorten terms (by, e.g., replacing the term (**ite** (**mod** 2 1 $\approx$ 0) 1 0) by 1), while the latter calls the simplification procedure only once. The latter turned out to be slightly faster. Therefore, we import the assumption in the local context, combine all the equations together, and finally, we discharge all assumptions and simplify the combined equation to $\bot$ at once.

We experimented on one arithmetic-heavy example stating the integer sequence of the form $x_{i+2} = |x_{i+1}| - x_i$ is periodic.

$$\begin{aligned}&\big(x_3 = |x_2| - x_1 \wedge x_4 = |x_3| - x_2 \wedge x_5 = |x_4| - x_3 \wedge\\ &\quad x_6 = |x_5| - x_4 \wedge x_7 = |x_6| - x_5 \wedge x_8 = |x_7| - x_6 \wedge\\ &\quad x_9 = |x_8| - x_7 \wedge x_{10} = |x_9| - x_8 \wedge x_{11} = |x_{10}| - x_9\big)\\ &\qquad \to x_1 = x_{10} \wedge x_2 = x_{11}\end{aligned}$$

The translation to the SMT-LIB format replaces the absolute values by if-then-else terms. The veriT proof involves 101 `la_generic` steps and reconstruction takes 3.6 s. If instead we use the `linarith` tactic to reconstruct the 101 arithmetic steps (the proof involves no strengthening), it takes 38.6 s. Hence, re-searching the coefficients can be a lot slower. Z3 produces a different proof that takes 4.3 s to reconstruct. We believe this is because the number of equalities in each step is smaller and, hence, fewer variables have to be eliminated.

To improve the efficiency of other rules we added a dedicated tactics sequence instead of relying on a generic tactic whenever possible. For example, the arithmetic rule la_disequality produces a theorem of the form $a \approx b \vee \neg a \leq b \vee \neg b \leq a$. The prototype reconstructed these steps using `linarith`, which is less efficient than applying the theorem.

3.2.2.4 Skolemization

The handling of skolemization is one of the most critical parts of the code, not only because it often occurs, but also because the reconstruction is not easy. Skolemization steps have the form $(\exists x.\ P\ x) \approx P\ X$ where X is defined as $(\epsilon\ x.\ P\ x)$. Since veriT can implicitly apply the symmetry of equalities, we might actually get $(\epsilon\ x.\ P'\ x)$ where P and P' are equal up to reordering of equalities. We have experimented with various ways to reconstruct such steps.

We now rely on the fact that the *name* of the dummy variable in the quantifier is unique within an equation and identical to the one in the choice term. Assume that, for example, the goal is $(\exists x_1\ x_2.\ (f\ x_1\ x_2) \approx 0) \approx ((f\ X_1\ X_2) \approx 0)$. We extract the names (x_1, x_2) from the equations and find the corresponding choice terms $X_1 = (\epsilon\ x_1.\ \exists x_2.\ 0 \approx (f\ x_1\ x_2))$ and $X_2 = (\epsilon\ x_2.\ (f\ X_1\ x_2) \approx 0)$. This also gives the order in which we have to replace the terms (first X_1, then X_2). Then we can fold the choice term in the term and discharge the proof obligation (first $((f\ x_1\ x_2) \approx 0) \approx (0 \approx (f\ x_1\ x_2))$, then $((f\ X_1\ x_2) \approx 0) \approx (0 \approx (f\ X_1\ x_2))$), *without* exploring the terms under the existential quantifier. The replacement can be done at several locations because not all choice terms have been folded into each other's definition.

Initially, we tried to add this process as a congruence rule or simplification rule to Isabelle/HOL, but the simplifier did not perform higher-order unification efficiently (it often unified with the wrong equations and only explored that branch).

In the case where everything is ordered correctly, this more general approach is slower than the previous solution, because inspecting terms and reordering equations is not necessary. Therefore, we first assume that no reordering occurred. If the goal is not solved, we extract the names.

While evaluating the reconstruction, we realized that defining choice terms can take a substantial amount of time. The solution is to reduce the term size by ensuring that veriT replaces quantified formulas by their associated choice definition, e.g., when $X_1 = (\epsilon\ x_1.\ x_1 \approx 0)$, writing $X_2 = (\epsilon\ x_2.\ x_2 \approx$

X_1) instead of $X_2 = (\epsilon\, x_2.\, x_2 \approx (\exists x_1.\, x_1 \approx 0))$. In the extreme example where we identified the issue, the number of characters in the definition of the Skolem term without sharing went down from over 3 million to around 800. The result is a large speedup, not only when converting to Isabelle/HOL terms, but also when defining the Skolem constants in veriT.

There are still some issues related to this. The replacement of terms by constants in veriT is restricted to syntactically equal terms. For example, instead of producing $X_1 = (\epsilon\, x_1.\, x_1 \approx 0)$, veriT could also produce the equivalent definition $X_1 = (\epsilon\, x_1.\, 0 \approx x_1)$. This term cannot used in X_2, because they are not syntactically equal. Such mismatches happen especially if equality is involved, since veriT applies symmetry without generating proofs. We cannot easily detect those cases in either veriT or Isabelle/HOL.

3.2.2.5 Selective Decoding of the First-order Encoding

Next, we consider an example of a rule that shows the interplay of the higher-order encoding and the reconstruction. To express function application, the encoding introduces the first-order function "*app*" and constants for encoded functions. The proof rule eq_congruent expresses congruence on a first-order function: $(t_1 \not\approx u_1) \vee \ldots \vee (t_n \not\approx u_n) \vee (f\ t_1 \ldots t_n) \approx (f\ u_1 \ldots u_n)$. With the encoding it can conclude $f \not\approx f' \vee x \not\approx x' \vee (app\ f\ x) \approx (app\ f'\ x')$. If the reconstruction unfolds the entire encoding, it builds the term $f \not\approx f' \vee x \not\approx x' \vee f\,x \approx f'\,x'$. It then identifies the functions and the function arguments and uses rewriting to prove that if $f \approx f'$ and $x \approx x'$, then $f\,x \approx f'\,x'$.

However, Isabelle/HOL β-reduces all terms implicitly, changing the term structure. Assume that f and f' are defined as $f := \lambda x.\, x \approx a$ and $f' := \lambda x.\, a \approx x$ in the example above. After unfolding all constructs that encode higher-order terms and after β-reduction, we get $(\lambda x.\, x \approx a) \not\approx (\lambda x.\, a \approx x') \vee (x \not\approx x') \vee (x \approx a) \approx (a \approx x')$. The reconstruction method cannot identify the functions and function arguments anymore.

Instead, the reconstruction method does not unfold the encoding anymore, including *app*. The steps can be reconstructed by treating *app* as an uninterpreted function and β-reduction is blocked. This eliminates the need for a special case to detect λ-abstractions. Such a case was used in the previous prototype, but the code was very involved and hard to test (such steps are rarely used).

3.2.2.6 Skipping Steps

The increased number of steps in the fine-grained proof format can slow down reconstruction. For example, since veriT distinguishes between the disjunction known by the SAT solver and the disjunction known by the theory part, it uses the or rule to convert between clauses and disjunctions. No such distinction is done in Isabelle/HOL. Hence, the rule only returns its premise. This rule accounts for more than one percent of the reconstruction time. A more involved example is skolemization of $(\exists x.\ P\ x)$. The proof generated by Z3 uses one step. The veriT solver generates *eight* steps – first renaming it to $(\exists x.\ P\ x) = (\exists v.\ P\ v)$ (with a subproof of at least 2 steps), then concluding the renaming to get $(\exists v.\ P\ v)$ (two steps), then $(\exists v.\ P\ v) = P\ (\epsilon\ v.\ P\ v)$ (with a subproof of at least 2 steps), and finally $P\ (\epsilon\ v.\ P\ v)$ (two steps).

To reduce the number of steps, our reconstruction skips two kinds of steps. First, it replaces every usage of the or rule by its only premise. Second, it skips the renaming of bound variables. The proof format treats $\forall x.\ P\ x$ and $\forall y.\ P\ y$ as two different terms and requires a detailed proof of the conversion. Isabelle/HOL, however, uses De Bruijn indices and variable names are irrelevant. Hence, we replace steps of the form $(\forall x.\ P\ x) \approx (\forall y.\ P\ y)$ by a single application of reflexivity. During preprocessing, veriT renames all bound variable to ensure that they are all distinct. Therefore, skipping renaming eliminates many steps.

We can also simplify the idiom

1.	▷	φ_1	(...)
2.	▷	$\varphi_1 \approx \varphi_2$	(...)
3.	▷	$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$	equiv_pos2
4.	▷	φ_2	(th_resolution 1, 2, 3)

veriT generates such steps for preprocessing, such as skolemization and variable renaming, that transforms a formula φ_1 into another formula φ_2. Step skipping replaces the equiv_pos2 and th_resolution steps by a single step which we replay using a specialized theorem.

On proof with quantifiers, step skipping can remove more than half of the steps – only four steps remain in the skolemization example above (where two are simply reflexivity). However, with step skipping the `smt` tactic is not an independent checker that confirms the validity of every single step in a proof.

3.2.3 Evaluation

During development we routinely tested our proof reconstruction to find bugs. As a side effect, we produced SMT-LIB files corresponding to invocations of `v-smt`. We measure the performance of veriT with various options on them and select five different strategies (Section 3.2.3.1). We also evaluate the repartition of the tactics used by Sledgehammer for preplay (Section 3.2.3.2), and the impact of the rules (Section 3.2.3.3).

We performed the strategy selection on a computer with two Intel Xeon Gold 6130 CPUs (32 cores, 64 threads) and 192 GiB of RAM. We performed Isabelle/HOL experiments with Isabelle/HOL version 2021 on a computer with two AMD EPYC 7702 CPUs (128 cores, 256 threads) and 2 TiB of RAM. Never versions of Isabelle/HOL have changed some details of the solver and tactic selection used by Sledgehammer.

3.2.3.1 Strategies

As we will discuss in Chapter V, veriT exposes a wide range of options to fine-tune the proof search that are combined into strategies. In order to select good strategies, we generate problems with Sledgehammer and use them to fine-tune veriT's search behavior. Generating problems also makes it possible to test and debug our reconstruction.

We test the reconstruction by using Isabelle/HOL's *Mirabelle* tool. It reads theories and automatically runs Sledgehammer [39 Section 4] on all proof steps. Sledgehammer calls various automatic provers (here the SMT solvers CVC4, veriT, and Z3 and the superposition prover E [86]) to *filter* facts and chooses the fastest tactic that can prove the goal. The tactic `smt` is used as a last resort.

To generate problems for tuning veriT, we use the theories from HOL-Library (an extended standard library containing various developments) and from the formalizations of Green's theorem [1, 2], the Prime Number Theorem [50], and the KBO ordering [19]. We configured Mirabelle such that Sledgehammer uses only veriT in the fact filtering step (see Chapter I). This produces SMT files for representative problems Isabelle/HOL users want to solve and a series of calls to `v-smt`. For failing `v-smt` calls three cases are possible: veriT does not find a proof, reconstruction times out, or reconstruction fails with an error. We improved the reconstruction such that it never fails or times out on the test theories.

Name	Options
del_insts	--index-sorts --index-fresh-sorts --ccfv-breadth --inst-deletion --index-SAT-triggers --inst-deletion-loops --inst-deletion-track-var
ccfv_SIG	--triggers-new --index-SIG --triggers-sel-rm-specific
ccfv_insts	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific --triggers-restrict-combine --inst-deletion-loops --index-SAT-triggers --inst-deletion-track-vars --ccfv-index=100000 --ccfv-index-full=1000 --inst-sorts-threshold=100000 --ematch-exp=10000000 --inst-deletion
best	--triggers-new --index-sorts --index-fresh-sorts --triggers-sel-rm-specific
default	(no option)

Table 3.1 Options corresponding to the different veriT strategies.

To find good strategies, we determine which problems are solved by several combinations of options within a two-second timeout. We then choose the strategy which solves the most benchmarks and three strategies which together solve the most benchmarks. To find those strategies we used the schedgen tool described in Chapter V. For comparison, we also keep the default strategy.

The strategies are shown in Table 3.1 and mostly differ in the instantiation technique (see Section 2.2.2). The strategy *del_insts* uses instance deletion [9] and uses a breadth-first algorithm to find conflicting instances. All other strategies rely on extended trigger inference [68]. The strategy *ccfv_SIG* uses a different indexing method for instantiation. It also restricts enumerative instantiation [80], because the options `--index-sorts` and `--index-fresh-sorts` are not used. The strategy *ccfv_insts* increases some thresholds. Finally, the strategy *best* uses a subset of the options used by the other strategies. Sledgehammer uses *best* for fact filtering.

We have also considered using a scheduler in Isabelle/HOL as used in the SMT competition. The advantage is that we do not need to select the strategy on the Isabelle/HOL side. However, it would make `v-smt` unreliable. A problem solved by only one strategy just before the end of its time slice can become unprovable on slower hardware. Issues with Z3 timeouts have been

reported by users on the Isabelle/HOL mailing list, e.g., due to an antivirus delaying the startup [63].

3.2.3.2 Improvements of Sledgehammer Results

To measure the performance of the v-smt tactic, we run Mirabelle on the full HOL-Library, the theory Prime Distribution Elementary (PDE) [49], an executable resolution prover (RP) [85], and the Simplex algorithm [70]. We extended Sledgehammer's proof preplay to try all veriT strategies and added instrumentation for the time of all tried tactics. Sledgehammer and automatic provers are mostly non-deterministic programs. To reduce the variance between the different Mirabelle runs, we use the deterministic MePo to select facts [73] instead of the better performing MaSh [66] that uses machine learning (and depends on previous runs). Furthermore, we do not use all CPU cores to reduce contention. We use the default timeouts of 30 seconds for the fact filtering and one second for the proof preplay. This is similar to the Judgment Day experiments [27]. The raw results are available in an online data archive [102].

Success Rate. Generally, users are not interested in which tactics are used to prove a goal, but in how often Sledgehammer succeeds. When the user invokes Sledgehammer, there are three possible outcomes: (i) Sledgehammer proposes a tactic that successfully proved the proof obligation during preplay, (ii) Sledgehammer proposes a tactic, but that tactic failed to prove the proof obligation (usually because of a timeout), or (iii) Sledgehammer fails. We define the success rate as the proportion of outcome (i) over the total number of Sledgehammer calls.

Table 3.2 gathers the results of running Sledgehammer on all goals of the selected formalizations and analyzing its outcome using different preplay configurations where only z-smt (the baseline) or both v-smt and z-smt are enabled. Any useful preplay tactic should increase the success rate (SR) by preplaying new proof hints provided by the fact-filter prover, reducing the preplay failure rate (PF). In contrast to the tests we use for development that are discussed above, we use different provers for the fact filtering step.

Let us consider the results when using CVC4 as fact-filter prover. The success rate of the baseline on the HOL-Library is 54.5% and its preplay failure rate is 1.5%. This means that CVC4 found a proof for 54.5% + 1.5% = 56% of the goals, but that Isabelle/HOL's proof tactics failed to preplay the

	HOL-Library (13 562 goals)				PNT (1715 goals)				RP (1658 goals)				Simplex (1982 goals)			
	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF	SR	OL_v	OL_z	PF
Fact-filter prover: CVC4																
z-smt	54.5		2.7	1.5	33.1		3.7	0.8	64.8		1.3	0.8	51.6		1.6	0.9
both	55.5	2.5	1.1	0.5	33.6	3.6	0.6	0.3	65.3	1.4	0.4	0.3	52.1	1.1	1.0	0.4
Fact-filter prover: E																
z-smt	55.5		1.1	1.7	36.0		0.3	1.7	61.7		0.7	1.2	49.8		1.4	0.7
both	56.0	0.8	0.7	1.3	36.4	0.6	0.1	1.3	62.1	0.9	0.2	0.8	49.9	0.3	1.3	0.5
Fact-filter prover: veriT																
z-smt	48.5		1.7	1.2	26.1		1.5	0.5	58.2		0.9	0.7	46.7		0.9	1.0
both	49.4	1.6	0.9	0.4	26.5	1.4	0.4	0.2	58.6	1.1	0.3	0.2	47.4	1.0	0.6	0.3
Fact-filter prover: Z3																
z-smt	50.8		2.5	0.8	27.9		2.7	0.4	60.4		0.8	0.7	48.3		0.9	0.3
both	51.3	1.9	1.1	0.3	28.2	2.5	0.5	0.1	60.9	1.1	0.1	0.2	48.4	0.4	0.6	0.2

Table 3.2 Outcome of Sledgehammer calls showing the total success rate (SR, higher is better) of one-liner proof preplay, the number of suggested v-smt (OL_v) and z-smt (OL_z) one-liners, and the number of preplay failures (PF, lower is better), in percentages of the unique goals.

proofs for 1.5% of goals. In such cases, Sledgehammer gives a proof hint to the user, which has to manually find a functioning proof. By enabling v-smt, the failure rate decreases by two thirds, from 1.5% to 0.5%, which directly increases the success rate by one percentage point: new cases where the burden of the proof is moved from the user to the proof assistant. The failure rate is reduced in similar proportions for PNT (63%), RP (63%), and Simplex (56%). For these formalizations, this improvement translates to a smaller increase of the success rate, because the baseline failure rate was smaller to begin with. The SMT solvers veriT and CVC4 both implement *conflict-based instantiation*. As a consequence, the addition of veriT to the smt tactic is especially useful when CVC4 is used in the fact-filtering step.

		Total	Shared Proofs						New Pr.
			Total =	OL_v	+	OL_z	+	OL_o	OL_v
		PR.	Time =	Time (Pr.)	+	Time (Pr.)	+	Time (Pr.)	Time (Pr.)
HOL-Library	z-smt	7409	250.1 =			85.0 (362)	+	165.1 (7047)	
	both	7545	212.6 =	27.9 (211)	+	19.6 (152)	+	165.1 (7047)	34.7 (135)
PNT	z-smt	569	33.4 =			14.8 (64)	+	18.5 (505)	
	both	577	28.4 =	7.7 (54)	+	2.1 (10)	+	18.5 (505)	3.4 (8)
RP	z-smt	1077	37.2 =			8.7 (22)	+	28.5 (1055)	
	both	1085	34.4 =	4.5 (16)	+	1.4 (6)	+	28.5 (1055)	2.2 (8)
Simpl.	z-smt	1024	42.8 =			6.7 (32)	+	36.0 (992)	
	both	1033	41.6 =	2.4 (13)	+	3.2 (19)	+	36.0 (992)	3.0 (9)

Table 3.3 Preplayed proofs (Pr.) and their execution time (s) when using CVC4 as fact-filter prover. Shared proofs are found with and without v-smt and new proofs are found only with v-smt. The proofs and their associated timings are categorized in one-liners using v-smt (OL_v), z-smt (OL_z), or any other Isabelle/HOL tactic (OL_o).

When using veriT or Z3 as fact-filter prover, a failure rate of zero could be expected, since the same SMT solvers are used for both fact filtering and preplaying. The observed failure rate can partly be explained by the much smaller timeout for preplay (1 s) than for fact filtering (30 s).

The choice between z-smt and v-smt is not clear-cut. Depending on the theories and on the back end, one performs better than the other. However, the best results are achieved when using both of them.

Overall, these results show that our proof reconstruction enables Sledgehammer to successfully preplay more proofs. With v-smt enabled, the weighted average failure rate decreases as follows: for CVC4, from 1.3% to 0.4%; for E, from 1.5% to 1.2%; for veriT, from 1.0% to 0.3%; and for Z3, from 0.7% to 0.3%. For the user, this means that the availability of v-smt as a proof preplay tactic increases the number of goals that can be fully automatically proved, by reducing the failure rate.

Saved time. Table 3.3 shows a different view on the same results. Instead of the raw success rate, it shows the time that is spent reconstructing proofs.

	Shared proofs				New proofs	
	OL_v		OL_z		OL_v	
	Time	Proof	Time	Proof	Time	Proof
No *best*	16.5	119	50.6	244	25.9	94
No *ccfv_SIG*	27.0	198	22.6	164	33.5	123
No *ccfv_threshold*	28.3	211	19.6	152	33.9	130
No *del_insts*	27.4	201	21.8	162	32.9	124
No *default*	27.9	207	20.1	156	33.8	134
Baseline	27.9	211	19.6	152	34.7	135

Table 3.4 Reconstruction time and number of solved goals when removing a single strategy (HOL-Library results only), using CVC4 as fact filter.

Using the baseline configuration, preplaying all formalizations takes a total of 250.1 + 33.4 + 37.2 + 42.8 = 363.5 seconds. When enabling v-smt, some calls to z-smt are replaced by faster v-smt calls and the reconstruction time decreases by 13% to 212.6 + 28.4 + 34.4 + 41.6 = 317 seconds. Note that the per-formalization improvement varies considerably: 15% for HOL-Library, 15% for PNT, 7.5% for RP, and 4.0% for Simplex.

For the user, this means that enabling v-smt as a proof preplay tactic may significantly reduce the verification time of their formalizations.

Impact of the Strategies. To understand the impact of the strategies on the Sledgehammer pipeline, we have also studied what happens if we remove a single veriT strategy from Sledgehammer (Table 3.4). The most important one is *best*, as it solves the highest number of problems. On the contrary, *default* is nearly entirely covered by the other strategies. *ccfv_SIG* and *del_insts* have a similar number where they are faster than Z3, but the latter has more unique goals and therefore, saves more time. Each strategy has some uniquely solved problems that cannot be reconstructed using any other. The results are similar for the other theories used in Table 3.3. Hence, even within the Sledgehammer pipeline the strategies retain their expected utility. The *best* strategy is the most useful and the *default* strategy, which was not optimized for Isabelle/HOL, is the least useful.

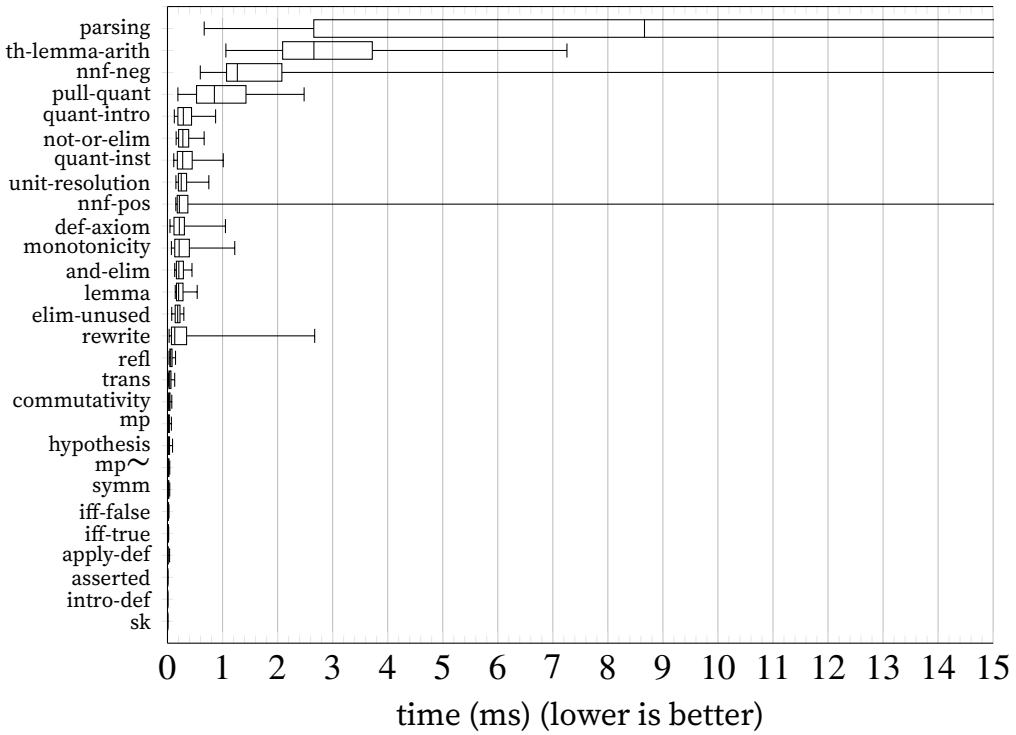


Figure 3.5 Timing of some of Z3's rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile. `nnf-neg`'s 95th percentile is 33 ms, `nnf-pos`'s is 87 ms, and `parsing`'s is 91 ms.

3.2.3.3 Speed of Reconstruction

To better understand what the key rules of our reconstruction are, we recorded the time used to reconstruct each rule and the time required by the solver over all calls attempted by Sledgehammer including the ones not selected. The reconstruction ratio (reconstruction over search time) shows how much slower reconstructing compared to finding a proof is. For the 25% of the proofs, Z3's concise format is better and the reconstruction is faster than proof finding (first quartile: 0.9 for `v-smt` vs. 0.1 for `z-smt`). The 99th percentile of the proofs (18.6 vs. 27.2) shows that veriT's detailed proof format reduces the number of slow proofs. The reconstruction is slower than finding proofs on average for both solvers.

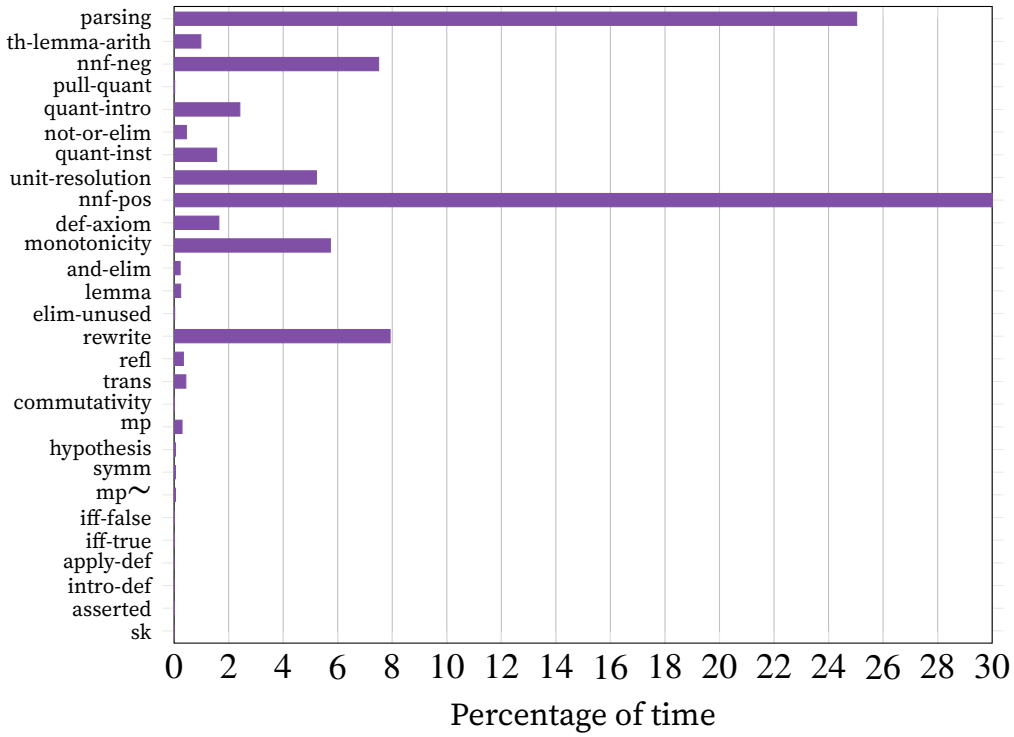


Figure 3.6 Total amount of time per rule for the SMT solver Z3. `nnf-neg` takes 39% of the reconstruction time.

Figure 3.7 shows the distribution of the time spent on all rules. We remove the slowest and fastest 5% of the applications, because garbage collection can trigger at any moment and even trivial rules can be slow. Figure 3.8 gives the sum of all reconstruction times over all proofs. We call parsing the time required to parse and convert the veriT proof into Isabelle/HOL terms.

Overall, there are two kinds of rules: (1) direct application of a sequence of theorems – e.g., equiv_pos2 corresponds to the theorem $\neg(\varphi \approx \psi) \vee \neg\varphi \vee \psi$ – and (2) calls to full-blown tactics – like qnt_cnf (Section 3.2.2.2).

First, direct application of theorems are usually fast, but they occur so often that the cumulative time is substantial. For example, cong only needs to unfold assumptions and apply reflexivity and symmetry of equality. However, it appears so often and sometimes on large terms that it is an important rule.

Second, rules which require full-blown tactics are the slowest rules. For the qnt_cnf rule (CNF under quantifiers, see Section 3.2.2.2), we have not written a specialized tactic but rely on Isabelle/HOL's tableau-based `blast`

tactic. This rule is rather slow, but is rarely used. It is similar to the rule la_generic: it is slow on average, but searching the coefficients takes even more time.

We can also see that the time required to check the simplification steps that were formerly combined into the connect_equiv rule is not significant anymore.

Furthermore, no tautological clauses with complementary literals are produced, although we could create artificial examples which produced such steps. The rules unary_minus_simplify, qnt_join, and la_tautology normalize terms by simplifying unary minuses, combining quantifiers, and handling inequalities where the variables can be eliminated. Their absence shows that terms written in Isabelle/HOL are already normalized.

We have performed the same experiments with the reconstruction of the SMT solver Z3. In contrast to veriT, we do not have the amount of time required for parsing. The results are shown in Figures 3.5 and 3.6. The rule distribution is very different. The nnf-neg and nnf-pos rules are the slowest rules and take a huge amount of time in the worst case. However, the coarser quantifier instantiation step is on average *faster* than the fine-grained steps produced by veriT. We suspect that reconstruction is faster because the rule, which is only an implication without choice terms, is easier to check. There is no problematic reordering of equalities.

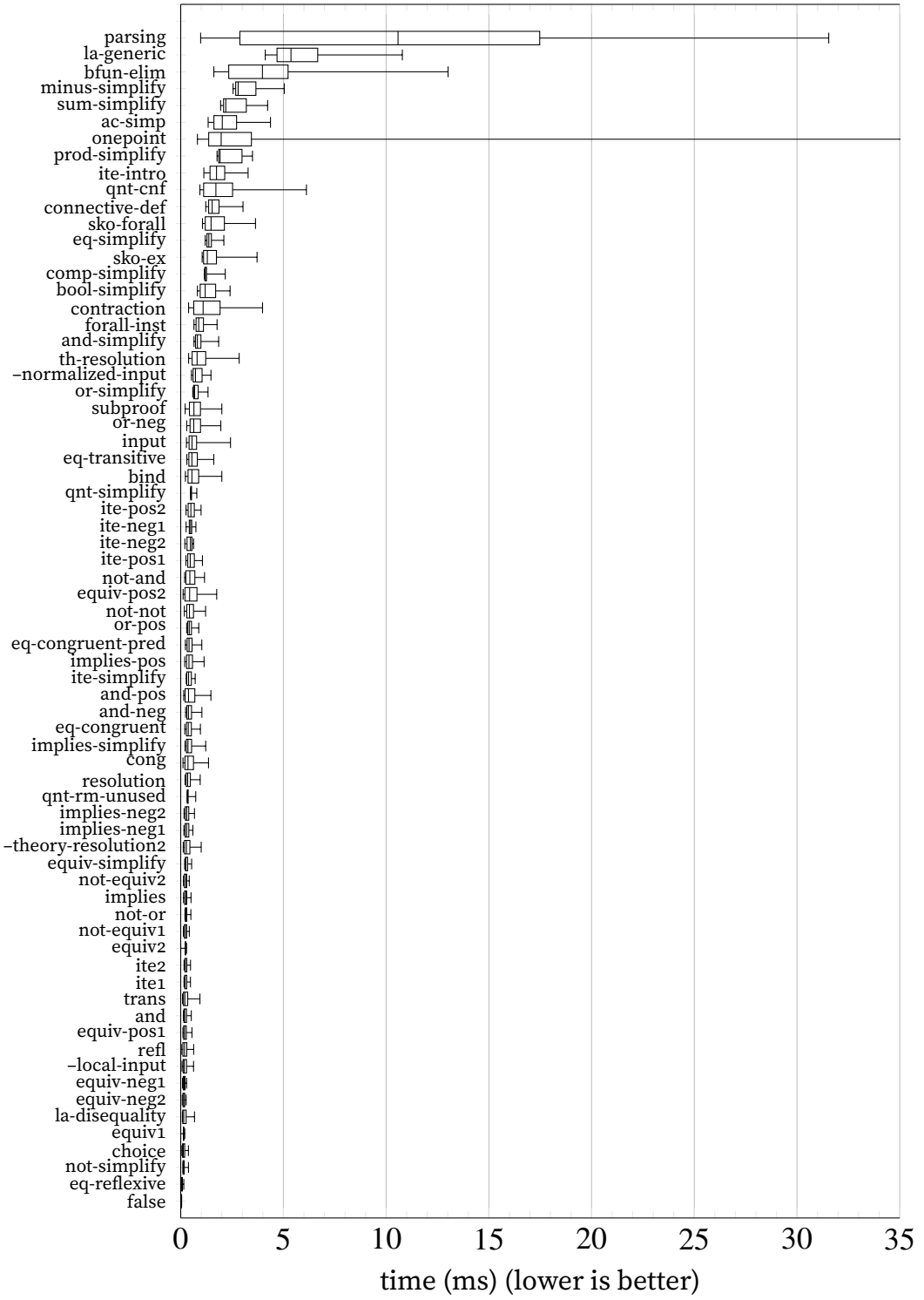


Figure 3.7 Timing of veriT rules sorted by median. From left to right, the lower whisker marks the 5th percentile, the lower box line the first quartile, the middle of the box the median, the upper box line the third quartile, and the upper whisker the 95th percentile.

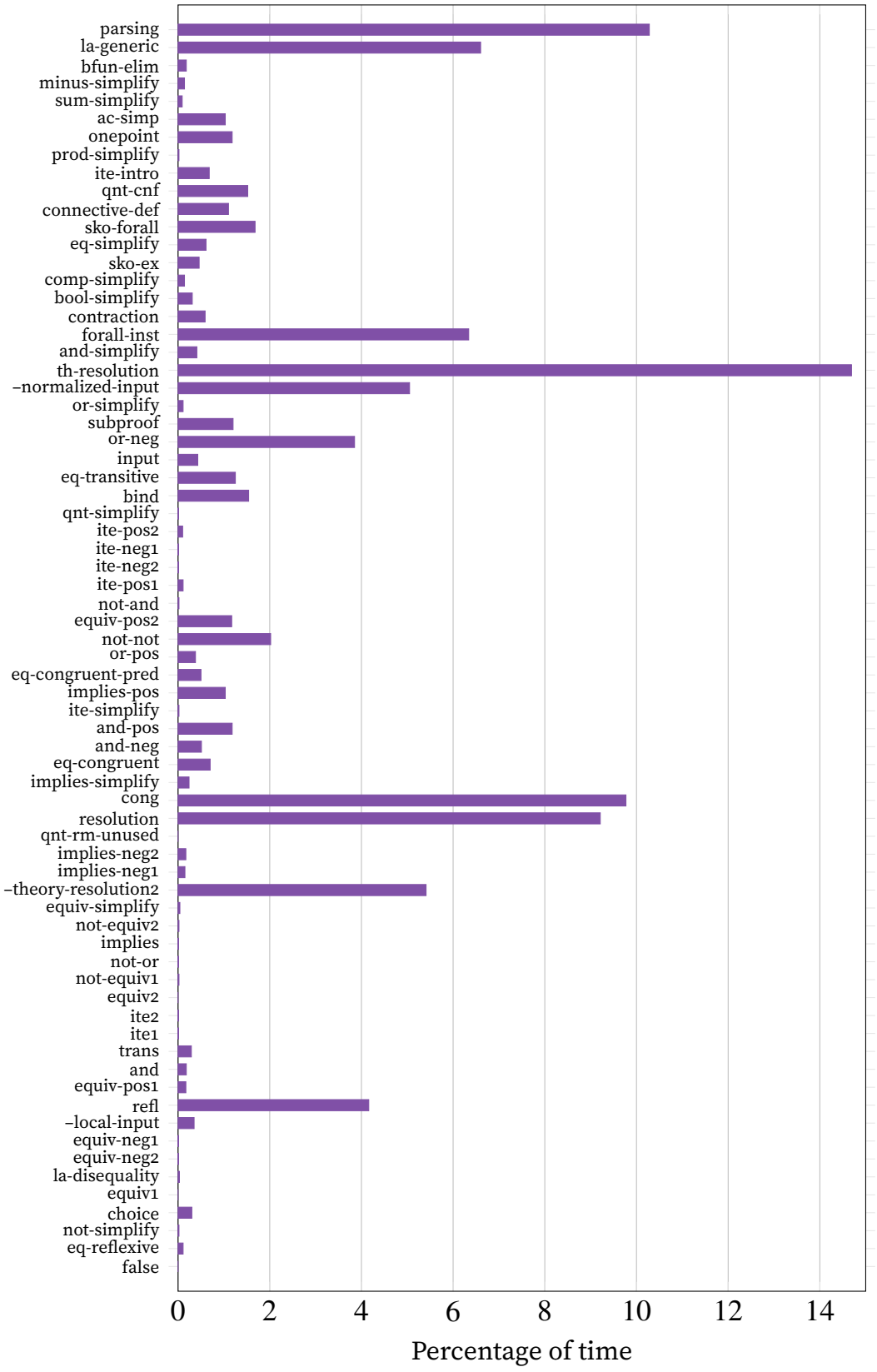


Figure 3.8 Percentage of the total time spent per rule for the SMT solver veriT.

3.3 Conclusion and Outlook

We will first, in Section 3.3.1, discuss other efforts related to generating proofs from SMT solvers and their reconstruction. Afterwards we will discuss the perspective for the reconstruction of veriT proofs in Isabelle/HOL in Section 3.3.2 and the perspective for the Alethe format in Section 3.3.3.

3.3.1 Related Work

The SMT solvers cvc5 [11], OpenSMT [62], SMTInterpol [33], veriT [31], and Z3 [47] produce proofs. Proofs from SMT solvers have also been used to find unsatisfiability cores [37] and interpolants [72]. They are also useful to debug the solver itself, since unsound steps often point to the origin of bugs.

With the version number change from CVC4, the SMT solver cvc5 got a reworked proof module [15]. The proof module of CVC4 does not record quantifier reasoning in the proof. The proof format of CVC4 also follows a different philosophy compared to veriT and Z3: it produces proofs in a logical framework with side conditions [91]. The output can contain programs to check certain rules. The proof format is flexible in some aspects and restrictive in others. The new proof module of cvc5 is designed to be very flexible and now supports quantifiers. It can produce proofs in multiple formats, beside the traditional logical framework with side conditions format, it also can produce proofs for the Lean proof assistant. Furthermore, there is experimental support to generate Alethe proofs. This support was added to generate proofs that can be reconstructed by Isabelle/HOL and SMTCoq.

OpenSMT uses different proof formats for the different components of solver such as propositional reasoning and theory reasoning [78]. Hence, a consumer must understand all formats relevant for them. OpenSMT does not support problems with quantifiers.

The proof format of the SMT solver SMTInterpol is very new [60]. It shares with Alethe the usage of SMT-LIB terms and explicit clauses, and with Z3 the usage of terms to express proof trees. In contrast to Alethe, steps do not always provide an explicit conclusion. Furthermore, the set of rules is much smaller. Hence, SMTInterpol proofs require less work to reconstruct, but solvers must perform some proof elaboration before printing. To support quantifiers the format uses rules that replace quantified variables with choice terms, similar to Alethe. Contrary to Alethe it does not feature a system to reason below binders. A proof checker for the format is available.

Proof reconstruction has been implemented into various systems, including CVC4 proofs in HOL Light [71], Z3 in HOL4 and Isabelle/HOL [28], and veriT [5] and CVC4 [53] in Coq. SMTCoq [5, 53] currently supports veriT's version 1 of the proof output which has different rules, does not support detailed skolemization rules, and is implemented in the 2016 version of veriT, which has worse performance. SMTCoq also supports bit vectors and arrays. There is an ongoing development effort to support Alethe proofs.

The reconstruction of Z3 proofs in HOL4 and Isabelle/HOL is one of the most advanced and well tested. It is regularly used by Isabelle/HOL users. The Z3 proof reconstruction succeeds in more than 90% of Sledgehammer benchmarks [24 Section 9] and is efficient (an older version of Z3 was used). Performance numbers are reported [26,28] not only for problems generated by proof assistants (including Isabelle/HOL), but also for preexisting SMT-LIB files from the SMT-LIB library. Unfortunately, the code to read SMT-LIB problems was never included in the standard Isabelle/HOL distribution and is now lost.[18]

The performance study by Böhme [26 Section 3.4] uses version 2.15 of Z3, whereas we use version 4.4.0, which is currently included with Isabelle/HOL. Since version 2.15, the proof format changed slightly (e.g., th-lemma-arith was introduced), fulfilling some of the wishes expressed by Böhme and Weber [28] to simplify reconstruction. Surprisingly, the nnf rules do not appear among the five rules that used the most runtime. Instead, the th-lemma and rewrite rules were the slowest. Similarly to veriT, the cong rule was among the most used (without accounting for the most time), but it does not appear in our Z3 tests.

The Dedukti [6] system takes a unique perspective to proofs intended to be consumed by software systems. It aims at being a lingua franca for proofs. At its core is a logical framework based on the λΠ-calculus modulo. Therefore, it is very expressive. There are modules to integrate Dedukti with multiple automated and interactive theorem provers. Furthermore, the Logipedia (http://logipedia.science) is a wiki of proofs that can be downloaded in multiple formats. It uses Dedukti internally. A hurdle for Dedukti and similar system is that its advanced type theory can be somewhat alien to many implementers of SMT solvers.

[18] private communication

3.3.2 Remaining Issues with the Reconstruction

Even though the reconstruction of Alethe proofs in Isabelle/HOL is now reliable, there are some avenues to improve it further.

Equality Ordering. One recurring challenge is the implicit reordering of equalities. It has a major impact on the reconstruction of skolemization steps and instantiation. Solving this problem is difficult. We implemented a prototypical version of veriT that normalizes terms as an explicit, proof producing preprocessing step. Unfortunately, this results in substantial performance regressions. Since the normalization was a separate step, the assumptions of veriT about the order of equalities would not hold at all times. This affects especially the preprocessing steps in subtle ways. We believe a more sustainable solution would be a post-processing of the proof that adds the missing normalization steps.

The new proof module of cvc5 can produce proofs for equality reordering. This was partially motivated by our experience. The cvc5 developers indicated in private conversation implementing this feature required a substantial amount of implementation effort.

Arithmetic The reconstruction of la_generic steps is not affected by the implicit reordering of equalities. Whenever an equality $t_1 \approx t_2$ is added to the arithmetic solver, this equality is always translated into the constraint $t_1 - t_2 \approx 0$. This makes the reconstruction of arithmetic steps easier and avoid backtracking to guess the direction of equalities. If we compare the amount of time spent on the reconstruction of arithmetic theorems, the Z3 reconstruction is faster. We believe that this due to a difference in the scheduling the arithmetic theory within the SMT solver, leading to simpler lemmas.

Another issue is the lia_generic rule. This rule is emitted by the branch-and-bound procedure in the case of inequalities containing variables with integer sort. Similar to the la_generic rule, it produces a conjunction of inequalities and equalities that is unsatisfiable in the theory of integers specifically. Unfortunately, reconstruction it is still an issue, because no detailed information is provided. We currently use the same reconstruction tactic as for Z3 which can fail even on simple equations like $3 \times p \not\approx 1$. Such simple equations, however, did not appear a single time in our tests on practical theories. Our tests also show that the lia_generic rule occur only once in our previous experiments, but never in the final version of them.

veriT and Isar Proofs for Sledgehammer. Sledgehammer has the ability to convert proofs generated by the solver to an Isar proof. Isabelle/HOL users use the Isar language to write proofs. Hence, it is optimized to be written and read by humans. The Isar proofs generated by Sledgehammer attempt to be close to human proofs. Consequently, the proof by contradiction is converted to a direct proof. Isar proof generated by Sledgehammer are potentially easer to understand and adapt by users.

One assumption made by Sledgehammer on the input proof is that skolemization are done in exactly one step and that the conclusion introduces the new Skolem constant that was never used before in the proof. This is obviously not correct for veriT's proof format. In the most simple case, we replace skolemization step that shows the equivalence with choice and simply keep the conclusion of the skolemization. This turns out to not be strong enough because skolemization can also happen in a lemma. In such cases, the Isar proof generated by Sledgehammer will most likely not work (and could even be nonsensical, like losing the conclusion). There is no simple solution to this problem, and fixing it remains future work.

3.3.3 Future Developments of Alethe

An outstanding aspect of the Alethe proof format is that support for it is developed in parallel by multiple systems. While the veriT integration is stable, we still occasionally find bugs. Furthermore, veriT does not aim at imposing its proof format on other systems. If proposals to improve Alethe are brought from other developers, we plan to implement those in veriT too.

The Alethe output by cvc5 is currently considered experimental by its developers. It is making good progress. It is planned to add support for the theory of bitvectors to Alethe as part of this effort.

Currently, there are three ongoing efforts on the Alethe consumer side. On the one hand, the reconstruction in Isabelle/HOL continues to be improved, especially with regard to the Alethe proofs generated by cvc5. On the other hand, there is ongoing work to update SMTCoq and to develop an independent proof checker.

The independent proof checker is developed in Rust.[19] It focuses on high-performance. In contrast to a proof-assistant-based reconstruction, the

[19] Available at https://github.com/ufmg-smite/alethe-proof-checker.

checker is not structured around a small, trusted kernel, and correct-by-construction extensions. Instead, the user must trust the implementation. Nevertheless, due to the usage of a safe programming language, modern development practices, and a completely independent code base, it achieves a substantial level of trustworthiness. The plan is to also extend the proof checker to perform transformations on an input proof. For example, it will be able to add proof steps for the implicit reordering of equalities performed by veriT.

A major downside of the parallel development of Alethe support is the challenge of keeping every implementation synchronized. For example, simple tweaks to the syntax that seem innocuous break the parsing of Alethe proofs in the consumers. The main way to overcome this is good collaboration between the developers and researchers.

Nevertheless, different SMT solvers have different requirements and capabilities in regard to the granularity of the proofs they can generate. To accommodate this without breaking interoperability we plan to design an annotation system for proof steps to express different levels of strictness for proof rules. This strictness can simplify the reconstruction, since less search is required. A good example of this is the trans rule that expresses transitivity. This rule has a list of equalities as premises, and the conclusion is an equality derived by transitivity. In principle, this rule can have three levels of "strictness":

1. The premises are ordered, and the equalities are correctly oriented (like in cvc5), e.g., $a \approx b$, $b \approx c$, and $c \approx d$ implies $a \approx d$.
2. The premises are ordered, but the equalities might not be correctly oriented (like in veriT), e.g., $b \approx a$, $c \approx b$, and $d \approx c$ implies $d \approx a$.
3. Neither are the premises ordered nor are the equalities oriented, e.g., $c \approx b$, $b \approx a$, and $d \approx c$ implies $d \approx a$.

The strictest variant is the easiest to reconstruct: a straightforward linear traversal of the premises suffices for checking. From the point of view of producing it from the solver, however, this version is the hardest to implement. This is due to implementations of the congruence closure decision procedure [42, 76] in SMT solvers being generally agnostic to the order of equalities, which can lead to implicit reordering that can be difficult to track.

For the trans rule, there are three different levels of strictness, but other proof rule could have multiple aspects that can vary in terms of strictness.

In any case, a consumer can implement support for strictness annotation by first implementing a (potentially slow) reconstruction for the least strict case. The addition of specialized routines for steps with strictness annotations is then an optimization step. Alternatively, an external tool could be used to translate steps to their strictest form. Such an annotation system requires some careful design considerations. Both the least strict variant and the strictest variant of each rule should be "natural" in some sense. The least strict variant must still be easy to reconstruct, and the strictest variant should not include pointless restrictions. Furthermore, the solvers should be able to use custom extensions.

Overall, the current rules should be cleaned up. For example, there are two resolution rules: resolution and th_resolution. Both rules are exactly the same, but are intended to be emitted by different components of the SMT solver: the first by the SAT solver, the second by a theory solver. While this is useful for debugging, this distinction should be expressed by an optional annotation. Furthermore, the definition of the resolution rule is very general. This is not out of necessity, but for safety. Since veriT emits this rule at many different places, we did not yet ensure it always adheres to a stricter definition of the resolution rule.

Another concrete example is quantifier instantiation. This is currently done by the forall_inst rule as described on page 61. This rule predates the fine-grained preprocessing rules that use contexts. During reconstruction, the user must perform a full substitution. Furthermore, the fine-grained rules for handling quantifiers can accompany simplifications in their subproof. Support for such simplification would be especially useful for quantifier instantiation. We propose the following alternative rule forall_inst'. It allows us to only instantiate some variables.

$$\text{i.}\ \frac{\Gamma, x_{l_1}, \ldots, x_{l_m}, \quad x_{k_1} \mapsto t_{k_1}, \ldots, x_{k_{n-m}} \mapsto t_{k_{n-m}} \rhd \varphi \approx \varphi' \qquad \text{(rule)}}{\text{j.}\ \Gamma \rhd \forall x_1, \ldots, x_n.\, \varphi \rightarrow \forall x_{l_1}, \ldots, x_{l_m}.\, \varphi'\ [(x_{k_1}, t_{k_1}), \ldots, (x_{k_{n-m}}, t_{k_{n-m}})] \qquad \text{forall\_inst'}}$$

where $m < n$ and $l_1, \ldots, l_m, k_1, \ldots, k_{n-m}$ are disjoint sublists of $1, \ldots, n$.

The conclusion of this rule differs from other rules that use contexts: it is an implication, not an equality. Therefore, to fully take advantage of forall_inst' other rules, such as the trans rule, should be adapted to work with

implications. After the proof rules are updated, they can be used to express, for example, instantiation of nested quantifiers precisely.

The rules used for preprocessing also deserve another look. Instead of rules that collect multiple transformations using rewriting, there should be small sets of precisely defined rules that can express all required rewriting steps. Furthermore, the expansion of user defined functions is currently unspecified. Of course, support for more theories is also on the wish list.

> are you acquainted with the mood of mind
> in which, if you were seated alone, and the
> cat licking its kitten on the rug before you,
> you would watch the operation so intently
> that puss's neglect of one ear would
> put you seriously out of temper?
>
> —Emily Brontë, Wuthering Heights

IV Improving Quantifier Simplification

Joint work with Pascal Fontaine This chapter was published at FroCoS 2021 [55]. I designed the presented method, implemented it, and performed the evaluation.

This chapter introduces a novel quantifier simplification technique. It is based on unification and is the result of many attempts to improve the quantifier reasoning capabilities of veriT, and SMT solvers in general. As discussed in this chapter, quantifier instantiation is very sensitive and methods that seem beneficial from a theoretical perspective turn out to be harmful in practice. Nevertheless, the method presented in this chapter works well. It is the result of studying an example, implementing the strictest necessary procedure to solve this example, and subsequent generalization. In the future the method can also be used as an inprocessing method to simplify existing formulas whenever new instances are generated. We also hope that it can serve as a first stepping stone for the wider use of resolution-based methods in SMT solving.

Since proof assistants use expressive logics, problems generated by them typically use quantifiers heavily. Furthermore, due to their interactive nature, proof assistants prefer solvers that respond quickly. As the empirical evaluation in Section 4.5 shows, the method presented here helps SMT solvers to solve problems quickly.

4.1 Reasoning With Quantifiers

As discussed in the introduction, the CDCL(T) calculus excels at handling quantifier-free problems with theories – SMT problems with thousands of assertions are frequent. For problems with quantifiers, SMT solvers use quantifier instantiation that generates new ground instances of the quantified formulas if the ground part of the problem can be satisfied.

When done fairly, this approach can be refutationally complete for many theories, and due to the strength of ground solving, it is also very powerful in practice. The main challenge is to find the right instances without misguiding or overwhelming the solver. Often, one can observe some kind of *butterfly effect*: if the instantiation methods are unlucky, the solver might be misguided to explore a large set of irrelevant instances and reach the solving timeout. In this regard, every strategy has its own strengths and weaknesses.

If the problem contains a quantified lemma that also occurs, for example, as an antecedent for another formula, common instantiation methods often

fail to quickly produce the right instances. This structure is quite typical of problems generated by interactive theorem provers.

Example 26
The following toy example illustrates this issue. It will illustrate various ideas throughout this chapter.

$$\forall x.\,((P\ x) \rightarrow (P\ (f\ x\ c))) \tag{4.1}$$

$$\forall y.\,((\forall z.\,((P\ z) \rightarrow (P\ (f\ z\ y)))) \rightarrow \neg(P\ y)) \tag{4.2}$$

$$(P\ c) \tag{4.3}$$

This problem is unsatisfiable: when y is set to c, assertion 4.1 occurs as the antecedent of the implication in assertion 4.2, so $\neg(P\ c)$ is a direct consequence of the first two assertions, in contradiction with the third. As described in Section 2.2.2, all major instantiation techniques fail to produce the correct instances for this problem in one round. Because SMT solvers typically only perform very limited preprocessing on quantified formulas and especially do not calculate a full clause normal form, the instantiation methods fail to recognize and exploit the fact that assertion 4.1 and the antecedent in assertion 4.2 are so similar. Since the instantiation methods do not produce the correct instances early, the SMT solver will need multiple instantiation rounds to solve the problem. This can lead to the butterfly effect mentioned above. Real-world examples are usually more complex. For example, there are often many ground terms which mislead the instantiation heuristics. Furthermore, the assertions in this example are horn clauses and could be handled by specialized reasoning. Practical problems, however, are not restricted to horn clauses.

For the SAT solver, quantified formulas are considered black boxes and are abstracted as propositional variables which occur as unit clauses in the propositional abstraction of the input formula. These propositional literals are generally of no value to the ground solver. We here make use of them to simplify larger formulas. To solve the example above we identify the occurrence of the unit assertion 4.1 within assertion 4.2. By using unification, we can eliminate this quantified subformula. The result after simplification is the ground formula $\neg(P\ c)$. After this formula is conjoined to the problem, the problem is trivially contradictory. In the general case, we use asserted quantified formulas to soundly simplify nested formulas and augment the

problem with the result. We propose multiple variants of the core procedure (Section 4.3).

So far, techniques inspired by resolution-based theorem provers are underrepresented in SMT solvers. Several systems such as DPLL(Γ) [46], DPLL(Γ+T) [29], and AVATAR [96] combine the superposition-based inference system of theorem provers with the CDCL(T) transition system on a fundamental level, but the combination is coarse – in those systems the two worlds work side by side in tandem. Instead, our unification-based method is a lightweight and easily implemented preprocessing technique that solves some concrete shortcoming of current instantiation techniques.

We implemented the method in the SMT solver veriT [31]. To ensure the process is fast, the implementation uses a standard term index and unification algorithm that is slightly extended to handle the presence of strongly quantified variables (Section 4.4).

The evaluation on SMT-LIB benchmarks shows that our technique enables veriT to solve benchmarks not solved with any strategy before. When applicable, the method often allows veriT to solve problems within a short timeout. The different variants of the simplification process are useful within a strategy schedule (Section 4.5).

The method also fits well into the Alethe proof framework. Although existing rules, as used by veriT, cannot express the simplification procedure, the extended proof rules for quantifier instantiation as discussed in Section 3.3.3 can capture the simplification very well. Section 4.6 discusses how this works.

4.1.1 Instantiation Fails

Let us first have a look how modern quantifier instantiation techniques handle our example. As discussed in Section 2.2 skolemization is applied during preprocessing, but only in a limited fashion and the preprocessed problem contains some disjuncts that start with a weak quantifier. To represent the "view" of the ground solver of such quantified disjuncts we use boxes as before.

Example 27
The preprocessor will not perform any operations on Example 26. The resulting clauses, illustrated with boxes, are $\boxed{1}$, $\boxed{2}$, and $(P\ c)$. The first two clause

are unit-boxes and both boxes will be abstracted to different propositional variables for the SAT solver.

We will now see how the common instantiation techniques fail to tackle Example 26. These techniques are presented in the order they are used by veriT: it first tries conflict-driven instantiation. If this fails, it will try trigger-based instantiation. Should this also produce no instances, it will fall back to enumerative instantiation. Model-based quantifier instantiation is not implemented by veriT: it is crucial for satisfiability, but veriT focuses mainly on proving unsatisfiability.

Conflict-Driven Instantiation. As discussed in Section 2.2.2, conflict-driven instantiation tries to find an instance that contradicts the ground model $\mathscr{M}$ in the theory of equality and uninterpreted functions (here the Core theory) [4, 13]. This method can find a contradicting instance of a clause $\psi_1 \vee \ldots \vee \psi_n$ by solving $\mathscr{G} \wedge \psi_1\theta \vDash_{\text{Core}} \bot, \ldots, \mathscr{G} \wedge \psi_n\theta \vDash_{\text{Core}} \bot$, but all ψ_i must be quantifier-free.

Since assertion 4.2 of Example 26 contains a quantifier, it cannot be instantiated by conflict-driven instantiation. Conflict driven instantiation also fails for assertion 4.1, because initially there is no ground formula that would be in conflict with an instance of $P\ (f\ x\ c)$. Even if the second assertion was skolemized, conflict-driven instantiation would fail: since there is no ground instance of the Skolem term, no conflicting instance can be found.

Trigger-Based Instantiation. Remember that this instantiation scheme works by matching *triggers* with the current ground model. In the case of Example 26, a trigger $(P\ x)$ on assertion 4.1 would produce the useless instance $(P\ c) \to (P\ (f\ c\ c))$ and a trigger $(P\ (f\ x\ c))$ initially cannot match anything. The trigger $(P\ y)$ on assertion 4.2 would produce the instance $(\forall z.\ ((P\ z) \to (P\ (f\ z\ c))) \to \neg(P\ c))$. This instance is a step towards solving the problem: the strong variable z is no longer below a quantifier and will be skolemized to create the formula $((P\ s_1) \to (P\ (f\ s_1\ c))) \to \neg(P\ c)$ where s_1 is a fresh constant. During the next instantiation round the trigger $(P\ x)$ on assertion 4.1 generates the instance $(P\ s_1) \to (P\ (f\ s_1\ c))$ which leads to the contradiction.

This technique is very sensitive to the availability of the right ground terms in the ground model. In the above example, if the formula contained

$\forall x.\,(P\ x)$ instead of $(P\ c)$, trigger-based instantiation would have been helpless.

Enumerative Instantiation. Enumerative instantiation lists ground instances without taking the ground model or the structure of the formula into account. For Example 26, enumerative instantiation also needs at least two rounds. First, the variable y of assertion 4.2 is instantiated with c. Then, after skolemization, assertion 4.1 can be instantiated with the new Skolem constant. Eventually, the cooperation of enumerative instantiation and the above techniques would succeed. However, in the presence of many ground terms of the same sort as c, enumerative instantiation might have needed a long time to find the right instance.

Model-Based Quantifier Instantiation. The remaining technique, model-based quantifier instantiation, extends heuristically the ground model $\mathscr{M}$ to a first-order interpretation $\mathfrak{I}$ and tests if this interpretation is a model. For Example 26, model-based quantifier instantiation fails to generate the right instances in one round for the same reason that trigger-based and enumerative instantiation fail: it could instantiate assertion 4.2 with c for y, but other rounds of instantiations will still be required to reach a contradiction.

4.2 Quantifier Simplification by Unification

The essence of our technique is to simplify boxes by replacing a quantified subformula of the box with the Boolean constant $\top$ or $\bot$. This can be done if the matrix of this quantified subformula can be unified with the matrix of a unit-box.

Example 28
On our running Example 26, the first assertion serves as a unit-box, whose matrix is unifiable with the matrix of the box in the second assertion. As a result, the quantified subformula can be reduced to the Boolean constant $\top$, for some instance of the second formula.

$$\frac{\forall x.\,(P\ x) \to (P\ (f\ x\ c)) \qquad \forall y.\,((\forall z.\,(P\ z) \to (P\ (f\ z\ y))) \to \neg(P\ y))}{\top \to \neg(P\ c)}$$

The rewriter simplifies the formula $\top \to \neg(P\ c)$ to $\neg(P\ c)$. Notice that, in this example, the variable z must be skolemized because its quantifier is strong.

The SUB rule (Section 4.2.1) formalizes this derivation. An SMT solver can use this rule to augment the problem with simplified formulas. It is carefully restricted to generate formulas that help the instantiation procedures, and has minimal drawbacks (Section 4.2.2). In Section 4.3 we propose several variants of the rule with different tradeoffs.

4.2.1 The Core Rule

The simplification by unification of subformulas (SUB) rule simplifies a box by replacing a quantified subformula with a Boolean constant. To be able to do so, the rule unifies the matrix of the subformula with a unit-box using a substitution. The Boolean constant depends on the polarities of the matrices: if they have the same polarity the subformula is replaced by $\top$; if they have different polarities it is replaced by $\bot$. The conclusion of the rule is the *pre-simplified* formula and will be fully simplified by the rewriter.

Definition 22 (SUB Rule)

$$\frac{\forall x_1, \ldots, x_n.\, \psi_1 \qquad \forall x_{n+1}, \ldots, x_m.\, \varphi[Q\bar{y}.\, \psi_2]}{\forall x_{k_1}, \ldots, x_{k_j}.\, \varphi[b]\theta}\ \text{SUB}$$

where $Q \in \{\exists, \forall\}$, the subformula $Q\bar{y}.\, \psi_2$ appears only below the outermost universal quantifier of φ, and θ is a substitution. The rule is subject to the conditions:

1. $\mathrm{trim}(\psi_1)\theta = \mathrm{trim}(\psi_2)\theta$, if $Q\bar{y}.\, \psi_2$ is weak;
2. $\mathrm{trim}(\psi_1)\theta = \mathrm{trim}(\mathrm{sko}(Q\bar{y}.\, \psi_2, x_{n+1} \ldots x_m))\theta$, if $Q\bar{y}.\, \psi_2$ is strong;
3. the bound variables of the conclusion $\{x_{k_1}, \ldots, x_{k_j}\}$ are exactly the variables $\mathrm{free}(\varphi[b]\theta)$;
4. $b = \top$ if $\mathrm{pol}(\psi_1) = \mathrm{pol}(\psi_2)$ and $b = \bot$ if $\mathrm{pol}(\psi_1) \neq \mathrm{pol}(\psi_2)$.

Example 29
The subformula $(\forall z.\, (P\ z) \to (P\ (f\ z\ y)))$ of assertion 4.2 $(\forall y.\, (\forall z.\, (P\ z) \to (P\ (f\ z\ y))) \to \neg(P\ y))$ occurs negatively. Since $Q = \forall$, the formula must be skolemized (Condition 2):

$$\mathrm{sko}(\forall z.\, (P\ z) \to (P\ (f\ z\ y)), y) = (P\ (s_1\ y)) \to (P\ (f\ (s_1\ y)\ y))$$

Hence, the unifier used in Example 28 is $\theta = [x \mapsto s_1(c), y \mapsto c]$. Using this unifier gives us the term $(\forall y.\, ((P\ (s_1\ y)) \to (P\ (f\ (s_1\ y)\ y))) \to \neg(P\ y))$, which is equisatisfiable to assertion 4.2 within the example. Furthermore, provided s_1 is fresh, satisfiability is preserved if, in a set containing assertion 4.2, this assertion is replaced by the skolemized formula.

Example 30
Condition 2 forces us to use skolemization. If we would ignore this condition, it would be possible to derive an unsatisfiable conclusion from satisfiable premises:

$$\frac{\forall x.\, (P\ x\ x) \qquad \forall y.\, \neg(\forall z.\, (P\ y\ (G\ z)))}{\neg\top}$$

In this case the skolemization used in condition 2 produces

$$\mathrm{sko}(\forall z.\, (P\ y\ (G\ z))\ y) = (P\ y\ (G\ (s_1\ y))).$$

This term is not unifiable with $(P\ x\ x)$. Hence, the rule is not be applicable if condition 2 is not ignored.

Example 31
This example shows that an application of the SUB rule can produce a formula that contains variables from both premises. In this example, the unifier $\theta = [y_1 \mapsto (G\ x), z \mapsto c]$ is used.

$$\frac{\forall x.\, (P\ (G\ x)\ c) \qquad \forall y_1, y_2.\, (\forall z.\, (P\ y_1\ z)) \wedge (P\ y_1\ y_2)}{\forall x, y_2.\, \top \wedge (P\ (G\ x)\ y_2)}\ \textsc{sub}$$

Example 32
The above examples were cases where $\mathrm{pol}(\psi_1) = \mathrm{pol}(\psi_2)$. This example illustrates the other case:

$$\frac{\forall x.\, \neg(P\ x\ x) \qquad \forall y.\, (G\ c) \wedge (\forall z.\, (P\ y\ z))}{(G\ c) \wedge \bot}\ \textsc{sub}$$

In this case, the rewriter will simplify the pre-simplified formula $(G\ c) \wedge \bot$ to $\bot$ and the SMT solver can directly deduce unsatisfiability.

The SUB rule allows us to simply combine and restrict skolemization, unification, and the replacement of subformulas with the appropriate constant. In the next section, we will see the role it has within an SMT solver. The

rule soundly combines these sound steps. First, it skolemizes the variables $\bar{y}$. Second, it applies the unifier θ. Now the subformula of ψ_1 corresponding to $Q\bar{y}.\,\psi_2$ in the SUB rule is equivalent to $\mathrm{trim}(\psi_1)\theta$ and is replaced with a Boolean constant. The constant is chosen appropriately according to the polarity of the formulas. This replacement is sound since $\psi_1\theta$ always holds. Overall, the SUB rule, together with the application of the rewriter, somewhat resembles unit resolution where $\forall x_1, \ldots, x_n.\,\psi_1$ is the unit clause. In the case of SMT solvers, however, φ might not be a clause. Furthermore, ψ_1 and ψ_2 will have the complex structure that is preserved from the input, since most currently used instantiation techniques have no advantage from applying full clausification and therefore don't apply it.

4.2.2 The Simplification within the SMT Solver

Since the SUB rule eliminates a quantified subformula, the conclusion is easier to handle for the SMT solvers. In general, however, the conclusion does not subsume the box that serves as the second premise. Hence, this box cannot simply be replaced by the conclusion. Instead, the problem must be augmented with the derived box. As the evaluations show, augmenting the problem still helps the SMT instantiation procedures to find the appropriate ground instances.

The pseudocode in Algorithm 4.1 shows the loop that augments the problem. It is executed after preprocessing finishes and before the ground solver starts. The procedure first iterates over the clauses in the preprocessed problem $\mathscr{P}'$ to build a set I of unit-boxes which can be used to simplify quantified subformulas. At the same time, this loop collects in a queue Q all clauses that contain boxes. Then the procedure takes a clause from the queue and tries to simplify one of its boxes. To do so, it uses the SUB rule. If this succeeds, the conclusion is the pre-simplified formula. The procedure then uses the rewriter to finish the simplification and the problem is augmented with the simplified formula by adding it conjunctively to the problem ($\mathscr{C}'$ is the new clause). If the simplified clause still contains a box, it is pushed onto the queue.

The procedure terminates since the queue Q will eventually be empty. Every iteration removes a clause from the queue and adds at most one new clause. When the test if the SUB rule can be applied fails no new clause is added. Otherwise, it adds a clause with fewer nested formulas that can serve

Require: A preprocessed problem $\mathscr{P}'$
Returns: The augmented preprocessed problems
  $I \leftarrow \emptyset$
  Q is an empty queue.
  for each clause $\mathscr{C}$ in $\mathscr{P}'$ **do**
    if $\mathscr{C}$ is unit-box with the box l **then**
      $I \leftarrow I \cup \{l\}$
    if $\mathscr{C}$ contains a box **then**
      push$(Q, \mathscr{C})$
    while Q is not empty **do**
      $l_1 \vee \ldots \vee l_n \leftarrow$ pop(Q)
      if there is $\psi \in I$ and a box l_i such that $\dfrac{\psi \quad l_i}{l'}\,_{sub}$ **then**
        $l' \leftarrow$ rewrite(l')
        $\mathscr{C}' \leftarrow l_1 \vee \ldots \vee l_{i-1} \vee l' \vee l_{i+1} \vee \ldots \vee l_n$
        append $\mathscr{C}'$ to $\mathscr{P}'$
        if $\mathscr{C}'$ contains a box and is not a unit-box **then**
          push$(Q, \mathscr{C}')$

Algorithm 4.1 The augmentation procedure.

as $Q\bar{y}.\,\psi_2$ in the SUB rule. Therefore, the SUB rule will eventually no longer apply to any box left in the clauses in Q.

The approach of augmenting the problem with derived, but new, formulas bears the risk that the instantiation procedures create more useless ground instances from the new formulas. To minimize this risk, the SUB rule is restricted to apply only when the result is likely to be helpful. First, the detection of subformulas which can be eliminated uses only unification instead of a more general approach. It does not take the properties of theory operators, such as the commutativity of $\vee$ and $\wedge$, into account. Since preprocessing preserves the structure of quantified formulas, unifiability can indicate the intention of the user. For example, the unit-box might be a lemma that is used within the box that is simplified. Second, the first premise must be a box. In principle it could also be a ground literal, but ground literals are already directly usable by the ground SMT solver. Third, the simplified subformula must start with a quantifier because the instantiation procedures struggle to instantiate the quantified subformula. One of

the variants described in the next section drops this restriction, but it is not as useful as the restricted rule.

4.3 Variants of Quantifier Simplification by Unification

As the experimental evaluation (Section 4.5) shows, the above variant of quantifier simplification by unification solves more instances at little cost. Nevertheless, we also developed several variants with different tradeoffs. We will call quantifier simplification by unification as presented so far the *standard variant* and will often drop the phrase *by unification* to avoid repetition.

Eager Simplification. Since quantified subformulas prevent instantiation procedures from creating the right instances quickly, the SUB rule is restricted to simplify only quantified subformulas. However, this restriction can be removed to generate more simplified formulas. This is the eager SUB rule.

Definition 23 (Eager SUB Rule)

$$\frac{\forall x_1, \ldots, x_n.\, \psi_1 \qquad \forall x_{n+1}, \ldots, x_m.\, \varphi[\psi_2]}{\forall x_{k_1}, \ldots, x_{k_j}.\, \varphi[b]\theta}\ \text{eager-SUB}$$

and all side conditions of SUB are changed to read ψ_2 in-place of $Q\bar{x}.\, \psi_2$, and the skolemization condition is ignored.

The eager SUB rule can be applied on any subformula not below an extra quantifier. On the one hand, this corresponds to deriving general consequences of unit-boxes in full first-order logic, but on the other hand, it will generate many more new formulas which potentially slow down or misguide the solver.

Solitary Variable Heuristic. To limit the potential downsides of eager simplification, we can limit the cases when the rule is applied: we apply the rule when it potentially removes a variable from the outermost quantifier of the second premise. The resulting formula will produce fewer potentially misleading instances. This is inspired by automated reasoning techniques that try to reduce the number of variables [34].

A variable is removed from the pre-simplified formula if it is *solitary*: it appears in the subformula ψ_2, but not in any other subformula of φ. Hence,

for example, in the case $\varphi = t_1 \vee \ldots \vee t_i \vee \ldots \vee t_n$ we apply the rule with $\psi_2 = t_i$ if there is a variable $x \in \mathrm{free}(t_i)$ such that $x \notin \mathrm{free}(t_1 \vee \ldots \vee t_{i-1} \vee t_{i+1} \vee \ldots \vee t_n)$.

Deletion of Second Premise. Another way to restrict the number of newly created instances is to delete the clause that contains the box used as the second premise of the SUB rule after it has been simplified. While this is no longer complete, it can guide the solver towards solving the refutation problem. Especially within a strategy schedule, this can be a valuable strategy.

This feature can be combined with the three variants. Overall, this results in six variants of quantifier simplification. The amount of clauses deleted depends on the activity of the simplification variant used. Especially in the case of eager simplification with deletion many input assertions will be deleted.

4.4 Implementation

Our implementation of quantifier simplification by unification in veriT uses a non-perfect discrimination tree as term index and a subsequent unifiability check (Section 4.4.1).

Both steps are amended to take strong variables into account without explicit skolemization and avoid the creation of unnecessary Skolem symbols. The SUB rule, as presented above, explicitly uses skolemization and, hence, creates fresh Skolem symbols every time it is applied.

The implementation also does not apply the simplification of clauses everywhere, but focuses on unit-boxes only: the queue Q will be populated only by unit-boxes. This simplifies the implementation, since we do not have to track which boxes of a clause were already simplified.

It indeed appears that in SMT-LIB benchmarks clauses with boxes are uncommon and quantified formulas are usually unit-boxes (e.g., quantifiers range over entire disjunctions). A prototype without this simplification did not perform better on these benchmarks than the simplified version.

4.4.1 Indexing and Unification without Skolemization

A key element to execute quantifier simplification, as shown in Algorithm 4.1, is the lookup of the unit-box $(\forall \bar{x}.\, \psi_1)$ from the index I. The trimmed matrix

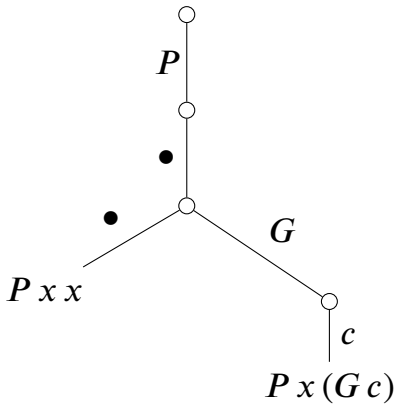


Figure 4.1 A discrimination tree indexing two terms.

of this box must be unifiable with the trimmed matrix of the quantified subformula $Q\bar{y}.\,\psi_2$. We use a term index to avoid a scan of the entire index for each candidate. Such an index allows the efficient lookup of terms unifiable with a query term. We use non-perfect discrimination trees [90]. Non-perfect means that the lookup is an over-approximation: some returned terms are not unifiable with the query term and must be removed by a full unification step. In comparison with perfect discrimination trees, non-perfect discrimination trees are easy to implement.

For each unit-box $(\forall\bar{x}.\,\psi_1)$ (of I in Algorithm 4.1) the index stores $\mathrm{trim}(\psi_1)$ together with $\mathrm{pol}(\psi_1)$. For each possible subformula $Q\bar{y}.\,\psi_2$ the implementation uses $\mathrm{trim}(\psi_2)$ as a query term and retrieves the unification candidates and their polarity. Afterwards, it performs a full unification to construct the substitution θ when possible. If, however, the quantifier of $Q\bar{y}.\,\psi_2$ is strong, the subformula should be skolemized. To take this into account, our implementation uses a slightly adapted lookup process.

Discrimination trees work similar to a string trie. Indexed and query terms are translated into strings by preorder traversal. Variables are replaced by a placeholder. For example, $P\,x\,(G\,c)$ becomes the string $[P, \bullet, G, c]$. The edges of the tree represent the individual characters of the indexed strings and the leaves store the index terms. Figure 4.1 shows a discrimination tree with the indexed terms $P\,x\,x$ and $P\,x\,(G\,c)$. During lookup the tree is traversed depth-first. If the query term or the indexed term contain variable placeholders, backtracking is used to collect all matches. Since variables are replaced by placeholders, the collected terms might not be

unifiable if the query or indexed term contain repeated variables. Hence, a full unification algorithm still has to be used, but most unification pairs that eventually fail are filtered out.

To handle variables that would be skolemized, the construction of the query string is slightly enhanced: it does not replace the implicitly skolemized variables $\bar{y}$ with the variable placeholder (•). Instead, the variables act like constants that can no longer match other constants and functions. Only variables in the indexed term can capture these. Since Skolem terms start with fresh function symbols which can never match indexed terms, this embeds skolemization on the fly into indexing. For example, the query term $P\ x\ (G\ y)$ matches both terms stored in the tree in Figure 4.1, but if y should be skolemized it will only match $P\ x\ x$. However, as Example 30 above shows, $P\ x\ x$ is also a false positive.

As the following example shows, the price to pay for this scheme is that more spurious terms can be retrieved from the index than when working with skolemized terms. Since anyway, full unification is used to build the substitution after the lookup, this is not a problem.

Example 33
Let us assume $(f\ x\ x)$ is stored as $[f, \bullet, \bullet]$ in the index. According to the description above, the term $(\exists z.\ (f\ y\ z))$ becomes the query string $[f, \bullet, z]$ which matches the stored term. When skolemized, however, it becomes $(f\ y\ (s_1\ y))$, which is not unifiable with $(f\ x\ x)$.

After the index returns a filtered set of possible premises $\mathrm{trim}(\psi_1)$ with their polarities $\mathrm{pol}(\psi_1)$ from the index I, the implementation must use full unification [84] to eliminate false positives and build the unifier θ. It has to solve the unification problem between $\mathrm{trim}(\psi_1)$ and $\mathrm{trim}(\psi_2)$, where $\mathrm{trim}(\psi_2)$ can contain variables that must be skolemized.

The unification algorithm iteratively builds a substitution θ by solving unification constraints of the form $t_1 = t_2$ by recursion on the term structure. If $f_1\,\bar{t} = f_2\,\bar{u}$, the unification fails if $f_1 \neq f_2$; otherwise it solves the new constraints $t_i\theta = u_i\theta$. If $x = t$, θ is extended with the substitution $[x \mapsto t]$ if x does not appear in t. The condition that x does not appear in t is the *occurs check*.

To handle skolemized variables during unification, our implementation of unification deviates from the standard version in two ways. First, similarly to the term index, it handles skolemized variables as constants. Second,

during the occurs check, it considers a skolemized variable as an occurrence of all the variables its Skolem term would depend on.

Example 34
In Example 30 the algorithm tries to unify $(P\ x\ x)$ with $(P\ y\ (G\ z))$ where skolemization would replace z with the Skolem term $(s_1\ y)$. Unification proceeds as follows:

1. to solve $(P\ x\ x) = (P\ y\ (G\ z))$ the constraints $x = y$ and $x = (G\ z)$ are added;
2. since $x = y$, the substitution is set to $\theta = [x \mapsto y]$;
3. to solve $x\theta = (G\ z)\theta$, that is, $y = (G\ z)$, the algorithm performs an occurs check for y in $(G\ z)$. Since z implicitly stands for $(s_1\ y)$, y appears in $(G\ z)$ and unification fails.

As a side effect of this, the resulting unifier θ cannot substitute a Skolem term into the quantified variables $x_{n+1}, \ldots, x_m$ of the box that is simplified. Therefore, the conclusion $\varphi[b]\theta$ is free of any Skolem terms, and no Skolem term must be constructed ever. Overall, restricting quantifier simplification by unification to not simplify formulas below multiple nested quantifiers allows for this elegant implementation.

4.5 Evaluation

This section presents an empirical evaluation of quantifier simplification by unification and its variants as implemented in veriT.[20] The default variant of quantifier simplification solves more benchmarks than the default configuration of veriT, while losing few benchmarks. This justifies the activation of our quantifier simplification method in the default configuration. Almost all other variants also solve more benchmarks than the default configuration. veriT exposes a wide range of options to fine-tune the instantiation module. A specific configuration is a *strategy*. Quantifier simplification solves benchmarks not solved by any veriT strategy without this technique (Section 4.5.1).

To fully benefit from the available strategies, veriT can use strategy schedules. We generated strategy schedules with and without quantifier simplification and evaluated their performance. Strategies with quantifier simplification are an integral component of the generated schedules and increase the

[20] The raw data is available on Zenodo [103].

number of solved benchmarks. They are especially useful for short timeouts (Section 4.5.2).

We performed the experiments on the benchmarks from the SMT-LIB benchmark release 2021 [17]. Since quantifier simplification is only relevant for first-order formulas, we used the SMT-LIB logics supported by veriT which use quantifiers, uninterpreted functions, or arrays. Those are the SMT-LIB logics UF, UFLRA, UFLIA, UFIDL, ALIA, AUFLIA, and AUFLIRA. Since veriT is can report only "unsat" when working on quantified problems, we removed benchmarks known to be satisfiable from the analysis.[21] Overall, the SMT-LIB contains 41 129 benchmarks using these logics. Of these, 1206 benchmarks are known to be satisfiable. This leaves 39 923 relevant benchmarks. We used the 2021.06-rmx release of veriT.

To interpret the numbers, the reader should keep in mind that veriT has no array solver. It treats the functions of the SMT-LIB theory of arrays as uninterpreted functions. Since veriT can only refute benchmarks, this approach is sound. Nevertheless, veriT can fail to solve easy benchmarks that require array reasoning.

All experiments have been performed on computers with one Intel Xeon Gold 5220 processors with 18 cores and 96 GiB RAM. We ran one instance of veriT per available core and used a memory limit of 6 GiB per instance.

4.5.1 Baseline Comparison

vs. Default (solves 31 690)	S	E	O	Sd	Ed	Od	Total
Solved	31 927	31 772	31 928	31 733	21 405	21 823	32 151
	+237	+82	**+238**	+43	−10 285	−9867	+461
Gained	282	**315**	285	291	115	255	475
Lost	**45**	233	47	248	10 400	10 122	14
vs. Virtual Best (solves 32 633)							
Gained	83	80	85	*86*	32	76	**125**

Table 4.1 Comparison with the default strategy and the virtual best solver on 39 923 benchmarks.

[21] Benchmarks known to be satisfiable can identify soundness problems. Hence, we included them in the experiments but removed them from the data.

Table 4.1 shows the number of benchmark solved within a timeout of 180 s in comparison to the default strategy. The standard variant of quantifier simplification by unification is denoted *S*, eager simplification is denoted *E*, and the solitary variable heuristic is denoted *O*. A suffix *d* denotes a variant that uses the deletion of the second premise feature. Benchmarks are "Gained" if they are not solved by the default strategy and "Lost" if they are solved by the default strategy, but not by the variant. The column "Total" reports the union of the benchmarks solved by all variants.

The standard variant shows a good improvement by solving 237 benchmarks more. Most other variants solve more benchmarks not solved by the default strategy, but also lose many more. Although the standard variant does not have the highest gain, the small loss justifies enabling it in the default strategy of veriT.

The huge number of lost benchmarks for the variants that use clause deletion with either eager simplification or the solitary variable heuristic is not surprising: since most input assertions can be simplified in some way, clause deletion removes much of the original problem. The result is often an unsolvable problem.

Compared to the union of benchmarks solved by *any* existing veriT strategy (Virtual Best), quantifier simplification by unification shows good improvement. We used a list of 43 strategies that are also used by veriT in the SMT competition.[22] The default configuration of veriT is on this list. Overall, the variants together are able to solve 125 benchmarks that veriT could not solve before. While eager simplification solves only 80 more, 18 of those are not solved by any other quantifier simplification variant. Here, the two variants with clause deletion that have a huge loss are somewhat redeemed: together they solve eight benchmarks not solved by the virtual best solver and the other quantifier simplification variants.

To perform the quantifier simplification, veriT does not need much time: for the standard variant, we measured a median runtime of 0.5 ms and mean of 3 ms.

Figure 4.2 shows the solving time of veriT with two variants of quantifier simplification by unification in comparison to the default configuration. Unsolved problems are clamped to a maximum run time of 180 s. This highlights a drawback of methods such as quantifier simplification by unification: even simplified terms can have a negative effect on the solving performance

[22] Competition web site: https://smt-comp.github.io/

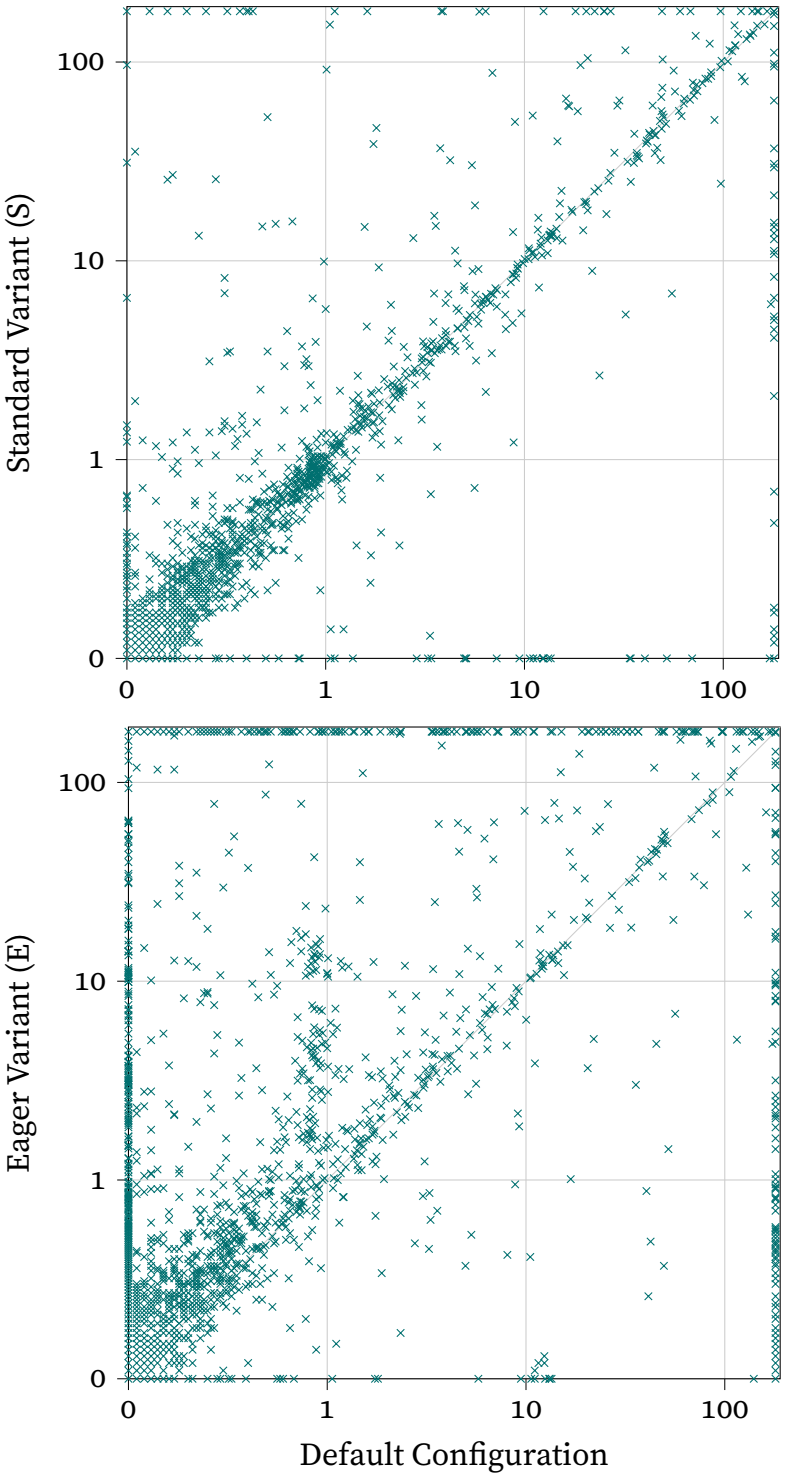


Figure 4.2 Comparison of solving time. Both axes are in seconds.

if they are added to the problem. For the successful standard variant of quantifier simplification by unification, there is a tendency for benchmarks to become slower. However, the data points at the bottom right of the plot show that many benchmarks that were not solved by the default configuration become trivial with quantifier simplification by unification. With eager simplification this effect is even more pronounced.

4.5.2 Strategy Scheduling

Since quantifier instantiation relies on heuristics, veriT exposes parameters that can be set by the user in a strategy. A specific choice of values for the exposed parameters is a strategy. Most benchmarks are solved by an appropriate strategy within a short timeout. Hence, it is sensible to execute many strategies for short time intervals one after another in a schedule.

To evaluate the quantifier simplification technique within a strategy schedule, we generated schedules with and without strategies extended with quantifier simplification. Schedules are automatically generated from a hand-crafted list of strategies and timeouts. An optimal schedule is the set of (timeout, strategy) pairs which solves the most benchmarks. We use the schedgen tool to generate such optimal schedules. Chapter V is dedicated to this tool. veriT itself uses the logic of the problem to select a schedule.

To build strategies with quantifier simplification, we picked six strategies from the strategy list of 43 strategies: the default strategy, the strategy that solved the most benchmarks overall, and four complementary strategies. The four complementary strategies were selected by finding a pair of strategies that together with the best strategy maximize the number of solved benchmarks. We searched for such a pair on all logics and on first-order logic with equality (UF) alone. We then extended these six strategies with the six variants of quantifier simplification. This resulted in 36 new strategies.

We generated schedules optimized for timeouts of 180 s and 24 s. The short 24 s timeout allows us to evaluate the value of quantifier simplification for applications such as interactive theorem provers, which require a short timeout. It corresponds to the timeout used by the SMT competition to evaluate solvers for this purpose. The longer 180 s timeout was arbitrarily chosen by us.

To generate the 24 s schedule we used the time slices 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20 seconds. To generate the 180 s schedule we used the time slices 1, 2, 4, 8, 16, 32, 48, 64, 80, 96, 112, and 128 seconds.

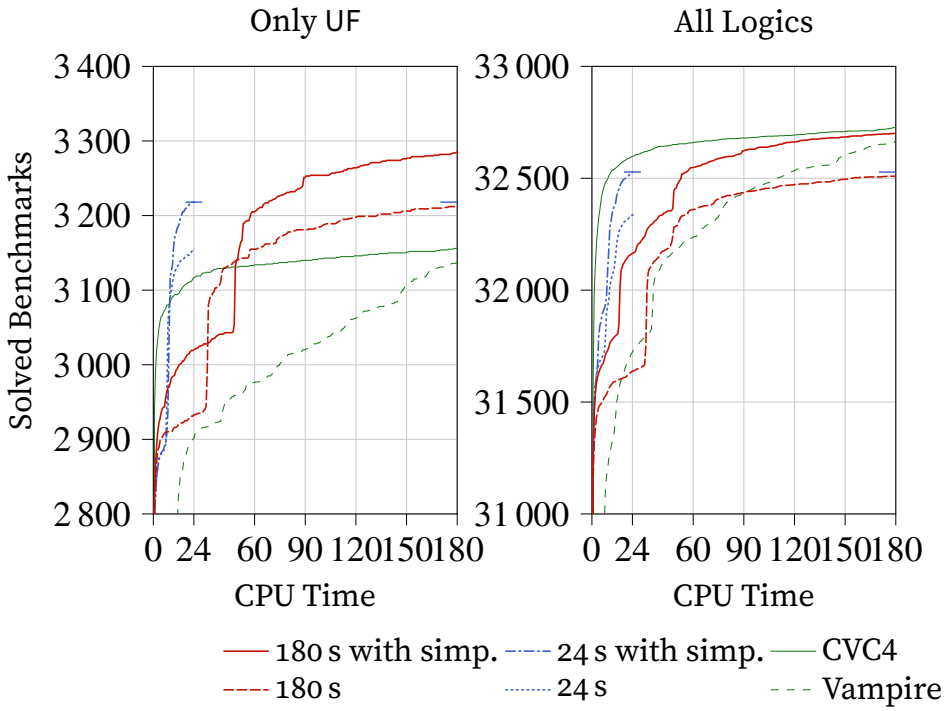


Figure 4.3 Cumulative distribution functions of different schedules on UF only and all logics.

Figure 4.3 shows the number of benchmarks solved within a time limit on UF alone and on all logics. On all logics, the schedule with quantifier simplification solved 193 benchmarks more after 24 s than the original 24 s schedule. For the 180 s timeout, the 180 s schedule with quantifier simplification solves 191 more than the one without it. The 24 s schedule with quantifier simplification solves 18 benchmarks more than the 180 s schedule without quantifier simplification after 180 s. Hence, quantifier simplification is very useful for short timeouts. Since the form of quantified lemmas that quantifier simplification by unification eliminates appear in problems generated by interactive theorem provers, it is especially useful for this application.

To provide context the plots contain the results of two other systems: the state-of-the-art SMT solver CVC4 and the superposition prover Vampire [65].

We used the official builds of version 1.8 of CVC4 and version 4.5.1 of Vampire. Vampire includes the SMT solver Z3, which aids theory reasoning. Since CVC4 has no scheduler optimized for 24 s or 180 s, we ran the default

strategy.[23] For Vampire we used the SMT-COMP scheduler with a timeout of 180 s.[24] We discarded all "satisfiable" results. Overall, CVC4 solves 70 benchmarks more than veriT with quantifier simplification after 24 s and 26 after 180 s. veriT with quantifier simplification after 180 s solves 595 benchmarks not solved by CVC4, of which 107 are also not solved by veriT without quantifier simplification.

Surprisingly, Vampire solves fewer benchmarks than any other system on UF. This is due to the nature of typical SMT benchmarks: they usually require little quantifier reasoning and are hence easier to solve for instantiation-based systems.[25] This confirms that restricted methods, such as quantifier simplification by unification, are useful for SMT problems. Recall that, contrary to veriT, CVC4 fully supports arrays. Implementing arrays seems like a priority for future versions of veriT.

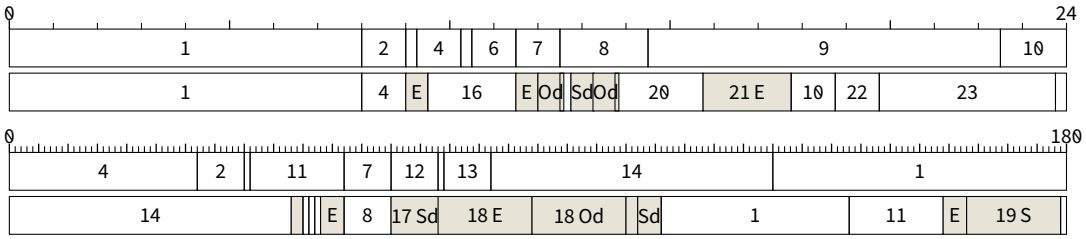


Figure 4.4 Visualization of optimized UF schedules. The bottom rows are the schedules with quantifier simplification. The numbers denote the base strategies.

Figure 4.4 visualizes the schedules for the logic UF. Gray cells are strategies that use quantifier simplification. Cells with the same number use the same base strategy. Some base strategies appear in both the schedules with and without quantifier simplification. Strategies with quantifier simplification tend to be used for shorter time slices than the variants without.

The majority of the new strategies are used during the first half of the schedule. For the 24 s schedule, the first strategy is extended with the standard variant of quantifier simplification and used for longer. The new 180 s

[23] Using: `-L smt2.6 --no-incremental --no-type-checking --no-interactive --full-saturate-quant`

[24] Using: `-t 180s -m 6000 --mode portfolio --schedule smtcomp --input_syntax smtlib2 -om smtcomp -p off`

[25] This has been confirmed to us by the Vampire team in conversations.

schedule starts with several strategies extended with quantifier simplification. Afterwards it uses the strategy 2 that the schedule without quantifier simplification uses right at the start.

Even aggressive simplification with deletion is used, albeit for a short timeout.

4.6 Proof Production

Quantifier simplification by unification integrates nicely into the Alethe calculus that is used by veriT to generate proofs. Unfortunately, it cannot be expressed by the currently existing rules related to quantifier handling. Hence, the current release of veriT cannot use quantifier simplification by unification when producing proofs. Nevertheless, the method would fit very well into the extended instantiation rules, as discussed in Section 3.3.3. This section discusses how.

A proof of the application of quantifier simplification by unification consists four components:

1. A proof of driving $\mathrm{trim}(\psi_1)\theta$ from $\mathrm{trim}(\psi_1)$ by applying the unifier θ.
2. A subproof starting with the application of θ to φ. In this subproof $\mathrm{trim}(\psi_2)\theta$ will be proven. It is syntactically equal to $\mathrm{trim}(\psi_1)\theta$.
3. A lift step, which introduces $\mathrm{trim}(\psi_1)\theta \approx \top$ into the subproof by using the conclusion of the first component. This enables the subproof to prove the pre-simplified formula.
4. An application of the proof producing rewriter to simplify the pre-simplified formula.

There is currently no rule which can express the lift step, but such a rule is a simple and natural addition. Furthermore, to prove the application of the substitution θ uses the fine-grained instantiation rule. It also uses bind steps to rename bound variables.

Example 35
Applying this process to the simplification of Example 28 we get the following Alethe proof.

1.		$\triangleright$	$\forall x.\,((P\ x) \to (P\ (f\ x\ c)))$	assume
2.		$\triangleright$	$\forall y.\,(\forall z.\,(P\ z) \to (P\ (f\ z\ y)) \to \neg(P\ y))$	assume
			...	
3.	$x \mapsto z$	$\triangleright$	$((P\ x) \to (P\ (f\ x\ c))) \approx ((P\ z) \to (P\ (f\ z\ c)))$	(rule)
4.		$\triangleright$	$\forall x.\,((P\ x) \to (P\ (f\ x\ c))) \approx \forall z.\,((P\ z) \to (P\ (f\ z\ c)))$	bind $[(x, z)]$
5.		$\triangleright$	$\forall z.\,(P\ z) \to (P\ (f\ z\ c))$	(... 1, 3)
6.	$y \mapsto c$	$\triangleright$	$\forall z.\,(P\ z) \to (P\ (f\ z\ c)) \approx \top$	(lift 5)
7.	$y \mapsto c$	$\triangleright$	$(P\ y) \approx (P\ c)$	(refl)
8.	$y \mapsto c$	$\triangleright$	$((\forall z.\,(P\ z) \to (P\ (f\ z\ c))) \to \neg(P\ y)) \approx (\top \to \neg(P\ c))$	(cong 6, 7)
9.		$\triangleright$	$\forall y.\,((\forall z.\,(P\ z) \to (P\ (f\ z\ c))) \to \neg(P\ y)) \approx (\top \to \neg(P\ c))$	forall_inst' $[(y, c)]$
10.		$\triangleright$	$\top \to \neg(P\ c)$	(... 2, 9)

Here, the lift rule lifts the result of the first substitution into the context of the second substitution. To be valid the variables in the context cannot be free in the lifted term.

4.7 Conclusion

This chapter presented a new unification-based simplification technique for instantiation-based SMT solvers. Its design is motivated by the limitations of modern instantiation methods. It is an efficient addition to the SMT solver. Problems where formulas can be simplified are often solved much faster, despite the method creating new quantified formulas. Quantifier simplification by unification is enabled by default in recent veriT releases.

We believe that the technique implemented here within veriT can be easily ported into any instantiation-based SMT solver, and we are confident

that it would also enable mainstream solvers to tackle problems beyond the reach of other current strategies.

Our method is a step towards using techniques inspired by resolution-based theorem provers within SMT solvers. It is currently only used as a preprocessing technique, but we plan to investigate novel quantifier instantiation techniques which can directly handle nested strong quantifiers.

Since proof assistants generate problems that use quantifiers extensively, the simplification method improves the utility of SMT solvers as proof assistant backends. The empirical evaluation showed that schedules with short timeouts profit almost as much from the new technique, as schedules with long timeouts. Therefore, the technique is useful in scenarios that require short timeouts – like proof assistants.

어디로 가고 싶나
바람따라 가고 싶네
어디로 가고 싶나
바람따라 가고 싶네
저 멀리 기차를 타고 갈까
저 멀리 버스를 타고 갈까
고속도로 달려 보네
불어라 봄바람아

—Kim Jung Mi, Blow Spring Breeze

V A Toolbox for Strategy Schedules

Parts of this chapter were published at PAAR 2021 [87].

Quantifier simplification by unification, as we saw in Section 4.3, exists in multiple variants which can be selected by the user. This is not extraordinary – most components of an SMT solver can be parameterized and fine-tuned. Selecting the right values for the parameters can be difficult. Often there is no clear best choice, and even if there is one, overall non-optimal choices might work better for certain types of problems. A specific choice of values for all exposed parameters is a *strategy*. It is often crucial to use the correct strategy to solve a problem within a given timeout.

An approach to this problem is for the solver to expose options to users that allow them to configure the used strategy. This does not solve the problem: defining the right strategy usually requires intimate knowledge of the inner workings of the solver. Furthermore, the developers of the SMT solver have to set a sensible default. This is not easy either: usually solver developers do not know up front the type of problems the solver will encounter. The default should normally also work well on a variety of problems. Overall, designing and using strategies is a subject that deserves some attention. Since for many problems there is a strategy that can solve the problem in a short time, it is natural to try multiple strategies on the problem. The easiest way to do this is to try strategies from a list one by one. Slightly more sophisticated is to prepare a list of strategies paired with a timeout: some strategies might be known to have diminishing returns if run for a longer time. We will call such a list a *schedule*.

The core of this chapter presents a method based on integer programming to find a strong schedule for a given set of strategies and benchmarks that is optimal with respect to the number of benchmarks solved. The SMT solver developer has to define a list of strategies and allowed time slices and has to record how much time each strategy uses to solve benchmarks from a training set. The method encodes this problem of finding the schedule that solves the most benchmarks within an overall timeout as an integer programming problem (Section 5.1 to 5.3). Overall, the generation procedure is designed to be easy to understand and avoid surprises.

This encoding is implemented in a toolbox called “schedgen” to generate schedules. It is implemented in Python (Section 5.4), has a simple user interface. Beside the core schedule generation tool, that toolbox also comes with auxiliary tools that can simulate and visualize schedules. The toolbox

allows users to generate simple, static schedules and to integrate them with their solver of choice. In Section 5.5 we see an evaluation of the practical usefulness of schedgen. Of particular interest to us is how well the generated schedules perform on benchmarks that are not used to generate the schedules. That is: do the generated schedules *generalize* to unseen benchmarks?

Section 5.6 discusses the wider picture how strategies can be used and where schedgen fits into this picture. On the one hand, the solver developer still has to provide the individual strategies. Hence, this section shows approaches to find good strategies. On the other hand, one could try to somehow analyze an input benchmark to determine up front which strategy could work well.

5.1 Integer Programming

To prepare for the next section, let us direct our attention to integer programming. It is widely used to express optimization problems. Solving an integer programming problem means finding an assignment to variables such that a linear combination of the variables is maximized and all linear inequalities from a given set are satisfied. If some variables must take integer values, while others allow real values, the problem is a *mixed* integer programming problem. Usually integer programming is presented in terms of linear arithmetic, but since we already have the machinery of many-sorted first-order logic, we will define integer programming in these terms.

Formally, a mixed integer programming problem is a tuple $(\mathscr{C}, o)$ where $\mathscr{C}$ is a set of formulas and o is a special term called the *objective function*. The formulas in $\mathscr{C}$ are formulas in the theory of linear real and integer arithmetic without Boolean connectives and quantifiers. They are called the *linear constraints*. The objective function is a term of sort **Real** or **Int**.

A solution to a mixed integer programming problem is an interpretation $\mathscr{I}$ such that $[\![t]\!]^{\mathscr{I}} = \mathsf{true}$ for all $t \in \mathscr{C}$ and there is no $\mathscr{I}' \neq \mathscr{I}$ such that $[\![o]\!]^{\mathscr{I}'}$ is greater than $[\![o]\!]^{\mathscr{I}}$. Hence, a solution $\mathscr{I}$ must maximize o.

Traditionally, the objective function o is written as a linear combination

$$(c_1 \times x_1) + (c_2 \times x_2) + \ldots + (c_n \times x_n)$$

of some variables $x_i : \sigma$ and constants $c_i : \mathsf{Int}$ where $\sigma \in \{\mathsf{Real}, \mathsf{Int}\}$. The linear constraints are traditionally written as equalities and inequalities in the form

$$(c_1 \times x_1) + \ldots + (c_n \times x_n) + c_{m+1} \bowtie (c_{n+1} \times x_{n+1}) + \ldots + (c_m \times x_m) + c_{m+2}$$

where $m \geq n$, the x_i and c_i are variables and constants just as in the goal function, and $\bowtie \in \{ =, \neq, <, >, \leq, \geq \}$. If the constant factors (c_{m+1} and c_{m+2}) are zero, they are usually omitted. Of course, it is necessary to add number conversion functions **to_real** and **to_int** to ensure the sorts of the subterms match. We will continue to omit those.

A special case of a variable of sort **Int** is a *binary* variable. An integer variable x is a binary variable if the problem contains the two constraints $x \geq 0$ and $x \leq 1$. Hence, a solution can only assign 0 or 1 to x. We will mark a binary variable with a dot: $\dot{x}$. Nevertheless, the sort of $\dot{x}$ is still **Int**. For simplicity, we will also not explicitly show the two constraints for each binary variable.

Since integer programming is a well established field, there is a lot of material available. One example is "*A Tutorial on Integer Programming*" by Cornuéjols, Trick, and Saltzman [35]. Furthermore, the online documentation of the PuLP library that is used by schedgen contains good introductory examples.[26]

5.2 Schedule Generation as Integer Programming

In this section, we will see how we can express the problem of finding an optimal schedule as a (mixed) integer programming problem. To do so, it is necessary to be a bit more precise about the notion of strategies, benchmarks, experiments, and schedules.

We have two finite, non-empty sets: the set S of strategies and the set B of benchmarks. Furthermore, we have a family of functions $(E_s \colon B \to \mathbb{R} \cup \{\infty\})_{s \in S}$ that represent the experiments. The value $E_s(b)$ is the time needed by the solver to solve the benchmark b when using the strategy s. If $E_s(b) = \infty$, the benchmark was not solved. To generate a schedule we also have to define the overall timeout $T > 0$ and a list $t_1, \ldots, t_n$ of allowed time slices. We assume that the list of time slices is in ascending order, all slices are positive, and no time slice appears twice (i.e., $0 < t_i < t_{i+1}$ for all $1 \leq i < n$). Furthermore, there is no value in having time slices that are longer than the overall timeout. Hence, $t_n \leq T$.

[26] The PuLP library is available at https://coin-or.github.io/pulp/

Within this environment, a *schedule* $\mathcal{S}$ is a finite set of pairs $(t, s) \in \mathcal{S}$ where $t \in \{t_1, \ldots, t_n\}$ and $s \in S$. The overall time used by the schedule must not exceed the total timeout:

$$\sum_{(t,s)\in\mathcal{S}} t \leq T.$$

The generated schedule should maximize the number of benchmarks solved within the overall timeout. Hence, we look for the schedule $\mathcal{S}$ such that

$$\left| \bigcup_{(t,s)\in\mathcal{S}} \left\{ b \mid b \in B \text{ and } E_s(b) \leq t \right\} \right|$$

is maximal. If the optimal solution for this objective function is found, the generated schedule is *optimal* in regard to the number of benchmarks solved.

One aspect is missing here: the order in which the strategies in $\mathcal{S}$ should be tried. This order can be determined in a second phase after the optimal set of (timeout, strategy) pairs has been calculated. This keeps the encoding simple. However, it is not obvious what measure should be optimized to find a good order. In Section 5.4 we will discuss different approaches to this problem.

Let $TS = \{t_1, \ldots, t_n\} \times S$ be the set of possible (time slice, strategy) pairs. We start modeling the optimization problem by defining a binary variable $\dot{x}_{(t,s)}$ for every $(t, s) \in TS$. If a $\dot{x}_{(t,s)}$ is 1, then the schedule runs the strategy s for the time t. We can now add our first constraint

$$\sum_{(t,s)\in TS} (t \times \dot{x}_{(t,s)}) \leq T.$$

This constraint guarantees that the execution time of the schedule is less than the total timeout. Since the $\dot{x}_{(t,s)}$ are binary variable the sum only adds the runtime of chosen pairs.

Since every benchmark solved by a strategy in a short timeout will also be solved by a longer timeout, it is pointless to pick one strategy for more than one time slice. To model this, we define the additional binary variables $\dot{x}_s$ expressing that the strategy s runs for any time slice. Additionally, we define one constraint for each such variable. For each $s \in S$ the constraint

$$\dot{x}_s = \sum_{1 \le i \le n} \dot{x}_{(t_i,s)}$$

expresses that each strategy can be picked at most once: since $\dot{x}_s$ is a binary variable, only one of the summands can be 1. This constraint eliminates corner cases. For example, if there is a schedule that solves all benchmarks well before the overall timeout, the integer programming solver could pick a strategy twice without changing the objective function.

With regard to solved benchmarks we will use two variables for each benchmark $b \in \boldsymbol{B}$. The binary variable $\dot{x}_b$ is intended to be 1 iff the benchmark b is solved by at least one picked (timeout, strategy) pair. That is, there is a $(t, s) \in \boldsymbol{TS}$ such that $\boldsymbol{E}_s(b) \le t$ and $\dot{x}_{(t,s)}$ is 1. To model this we use an auxiliary *integer* variable x'_b that counts the number of times b is solved. This is implemented for each x'_b by the constraint

$$x'_b = \sum_{\dot{x} \in X_b} \dot{x} \text{ with } X_b := \left\{ \dot{x}_{(t,s)} \mid (t, s) \in \boldsymbol{TS} \text{ and } \boldsymbol{E}_s(b) \le t \right\}.$$

The following two constraints force $\dot{x}_b$ to 1 iff $x'_b \ge 1$.

$$\dot{x}_b |X_b| \ge x'_b$$
$$\dot{x}_b \le x'_b + 0.5.$$

If x'_b is not 0, then the first constraint ensures that $\dot{x}_b$ has to be 1, if x'_b is 0, then the second constraint ensures that $\dot{x}_b$ is 0 too. Since x_b is a binary variable, there is no other value smaller than 0.5.

Defining the objective function is now easy. It is just the sum

$$\sum_{b \in B} \dot{x}_b.$$

To extract the schedule $\mathcal{S}$ from a solution of this encoding, we can collect the $\dot{x}_{(t,s)}$ that are set to 1. Due to the constraints, no two $\dot{x}_{(t_1,s)}$, $\dot{x}_{(t_2,s)}$ with $t_1 \neq t_2$ are ever 1.

A solution to this integer program maximizes the number of benchmarks solved within the overall timeout. Every strategy is assigned at most one of the allowed time slices. Since the time slices are predefined, this solves a limited form of the general optimal schedule problem [100] which is NP-hard. Our restricted version is also a variant of the knapsack problem: we want to

maximize the value of the items (the number of solved benchmarks by the strategies) while respecting the weight limit (the timeout). Therefore, it is still NP-hard.

5.3 Strategy Order and Combined Schedules

The solution of the integer program as discussed above does not give an order of the (timeout, strategy) pairs. The goal for selecting pairs is clear: solve as many problems as possible. There is also an intuitive goal for the order: to minimize the sum of solving times of the benchmarks solved by the schedule. Indeed, if we want to generate a schedule that performs as well as possible at the SMT competition, we would have to find the order that minimizes the sum of all solving times. The single-query track of the competition ranks the solvers by the number of benchmarks solved within the overall timeout (20 min), but the sum of solving times serves as the tie breaker [14 Section 7.1].

This is another optimization problem that is further complicated by the unpredictable behavior of the solver on unsolved benchmarks. The solver might either run until terminated at the timeout, give up, or even crash. Since veriT uses quantifier instantiation, it often gives up when no new ground instances are generated because the input problem is satisfiable. In this case, no strategy can succeed, but all will be tried. Currently, the implementation of the optimization toolbox does not provide a procedure to compute this order. We tried an approach based on dynamic programming, but it was too slow in practice.

Furthermore, this order is not necessary the best for all application. For example, an application could interrupt ordered schedules prematurely. To see why minimizing the sum of solving times is not the best approach in this situation, consider two strategies a and b such that a solves three benchmarks after 2 s and b solves only one benchmark after 1 s. The sum of solving times of the four benchmarks of the list $[(2, a), (1, b)]$ is 9 s. The sum for the list $[(1, b), (2, a)]$ is 10 s. However, if the first list is interrupted after one second, no benchmark is solved. If the second list is interrupted after one second, one benchmark is solved.

Alternatively, one might sort the (timeout, strategy) pairs by increasing timeout. The result is a schedule that tries short strategies in quick succession before using longer strategies. However, it might be that strategies that are only used with a short timeout are specialized to a small number

of benchmarks and the more general strategies are used with longer timeouts. Furthermore, since the time needed to solve benchmarks is usually not evenly distributed, even a strategy used with a long timeout might solve many benchmarks very fast.

An alternative approach is to pick the pair that solves the most benchmarks not solved so far. This *best effort* order has the benefit that if the schedule is not used for the total timeout, the number of benchmarks not solved is minimized if the schedule is interrupted between pairs. The downside of this approach, however, is that the schedule would use a pair with a long timeout before using two pairs with short timeouts that cumulatively solve more benchmarks.

5.3.1 An Improved Best Effort Order

As we saw above, the order of a strong schedule should take the number of solved benchmarks, the time it takes to solve them, and the timeout of each pair into account. We can modify the best effort order to achieve this. Instead of picking the pair that solves the most benchmarks, we pick the pair that has the minimal estimated cost. To estimate the cost of a pair we calculate the sum of solving times. For benchmarks solved by the pair this is straightforward. For the other benchmarks an approximation is needed. Our approximation is the solving time used by the virtual best solver formed by the other pairs in the schedule. Here, the virtual best solver is constructed from all available strategies. The virtual best solver of a schedule $\mathcal{S}$ is the function $E_{\mathcal{S}}\colon B \to \mathbb{R} \cup \{\infty\}$ such that $E_{\mathcal{S}}(b) = \min(\{E_s(b) \mid (t,s) \in \mathcal{S} \wedge E_s(b) \leq t\} \cup \{\infty\})$. Based on this definition, the cost estimate for a pair $(t,s) \in \mathcal{S}$ formally is

$$c_{(t,s)} = \sum_{b \in \{b \mid E_s(b) < t\}} E_s(b) + \sum_{b \in \{b \mid E_s(b) > t \wedge E_{\mathcal{S}'}(b) < \infty\}} \left(t + E_{\mathcal{S}'}(b)\right).$$

where $\mathcal{S}' = \mathcal{S} \setminus (s,t)$.

The ordered schedule is constructed by removing iteratively the pair (t,s) with the smallest cost estimate $c_{(t,s)}$ from $\mathcal{S}$ until no pair is left. The resulting schedule will usually first use pairs with short timeout. Longer timeouts will only be chosen if the benchmarks solved by this strategy offset the delay to solving the other benchmarks. Since a schedule typically contains less than a hundred pairs, ordering a schedule with this procedure is fast.

5.3.2 Combining Schedules

It is also possible to target multiple predefined timeouts with one schedule. For example, at the SMT competition, solvers compete with both a timeout of 20 min and a timeout of 24 s. A simple approach to generate a schedule that work well in this setting is to perform multiple rounds of optimization: let T_1 and T_2 be two timeouts, such that $T_1 < T_2$ and let $\mathcal{S}_1$ be the optimal schedule for the timeout T_1. We can now define a new set of benchmarks B' that contains exactly the benchmarks from B that are not solved by the schedule $\mathcal{S}_1$ within the timeout T_1. Next, we can calculate another schedule $\mathcal{S}_2$ that solves the most benchmarks from B' within the timeout $T_2 - T_1$. Both rounds of optimization can use a different set of allowed time slices. To build the ordered joint schedule, we just search for the best order for both schedules independently and concatenate the resulting lists.

A small downside of this approach is that some strategies appear in both schedules and the solver has to perform some repeated work. This repeated work can be avoided for one strategy. Assume that there is a repeated strategy s. Hence, there is a pair $(s, t_1) \in \mathcal{S}_1$ such that $(s, t_2) \in \mathcal{S}_2$ for some t_2. In this case $t_2 > t_1$ since otherwise (s, t_2) would solve no benchmarks not already solved by $\mathcal{S}_1$. If we remove (s, t_1) from the first schedule $\mathcal{S}_1$, the schedule solves fewer benchmarks, but only runs for the time $T_1 - t_1$. To get those benchmarks back, we move (s, t_2) to the beginning of the schedule $\mathcal{S}_2$. In the concatenated schedule, s is used for the time t_1 before T_1 passes and is used for the time t_2 overall.

5.4 The schedgen Toolbox

We implemented the schedule generation procedures described above as part of a toolbox called “schedgen”. Beside optimization, the toolbox also contains some other tools that are useful when working with strategy schedules, such as a visualizer and simulator. This section provides an introduction to the toolbox. It also describes important implementation details of the toolbox. The source code for the toolbox is available under an open source license (https://gitlab.uliege.be/verit/schedgen/).

The schedgen toolbox is implemented in the Python programming language and uses the PuLP [74] package to express and solve the linear programming problems. The PuLP package can use multiple linear programming

solvers as backend, but the default solver is good enough for the linear programming problems arising from our use case. The default solver is the open source solver Cbc.[27] Both the PuLP and Cbc project are developed by the COIN-OR foundation, which develops open source software for operations research.[28] To store the experiment results schedgen uses the Pandas library.[29] This library provides a flexible data structure to work with tables.

All tools in the schedgen toolbox support the filtering of benchmarks by SMT-LIB logic. Since the SMT-LIB benchmark library is big (around 100 GiB), it is often not installed on the computer that is used for schedule generation. In this case, the logic cannot be determined by parsing the benchmark file. Instead, usually the first component of the benchmark path is used. For example, a benchmark QF_UF/test.smt2 will have the logic QF_UF assigned. This matches the typical folder structure used by the SMT-LIB benchmark suite. In fact, it is not necessary to use the SMT-LIB logic for filtering. It is also possible to use another string that occurs in the benchmark name. This allows the user to create custom subsets of benchmarks.

The schedgen toolbox is a standard Python package. It has been tested with Python 3.10. To install the package it is enough to execute

```
$ python setup.py install
```

This will install the following tools that are part of the schedgen toolbox on the user systems.

`schedgen-optimize.py` is the main tool that performs the schedule generation. It takes the name of a folder that contains the experiments, a list of time slices, and the total timeout. After the generation is finished the result is written as a schedule into a CSV file. It is also possible to provide an existing scheduler to build a joint schedule. The benchmarks used to generate the schedule can be filtered as described above.

`schedgen-finalize.py` takes CSV files describing schedules, as generated by `schedgen-optimize.py`, and produces a shell script that runs a solver according to the schedules. The user can provide a template for the script and change

[27] Available on https://github.com/coin-or/Cbc.
[28] For more information see https://www.coin-or.org/
[29] See https://pandas.pydata.org/.

various parameters used during generation. The default script can pick schedules by SMT-LIB logic.

`schedgen-simulate.py` can be used to predict the outcome of running a scheduler. To do so the user has to provide experiments for the strategies included in the schedule, but can use different benchmarks. Furthermore, it is possible to add noise to the simulation.

`schedgen-query.py` is a small tool that produces lists of benchmarks for debugging and analysis purposes. For example, it can list the union of the benchmarks solved by some strategies, or find the benchmarks from this list that a specific schedule cannot solve.

`schedgen-visualize.py` can be used to visualize one or multiple schedules. It generates a bar plot that represents a schedule visually.

Each of those tools can generate an extensive documentation of the options it exposes via the standard `--help` option.

The tools accept experimental data in two formats: the GridTPT format and a simple CSV format. The GridTPT system [30] is a platform for testing SMT solvers and other theorem provers on grid computers. Its file format is based on CSV, but extended with a header that stores information about the experiment. For every strategy one file is used that contains the results for all benchmarks. The header also contains the command line options, i.e., the strategy used for the experiment. Furthermore, the header contains information such as the solver, when it was executed, and what timeout was used. Since the GridTPT format uses one file per strategy, the user has to declare a folder that contains all data files. The schedgen input parser searches recursively for compatible files in this folder.

The simple CSV format collects all results within one file. The data file contains five columns and header that stores the column names. Columns are separated by a literal ";". The columns can be given in any order. The parser uses the header line to determine the order of the columns. The five columns are:

- `benchmark` contains the filename of the benchmark.
- `logic` stores the SMT-LIB logic of the benchmark or any other string that should be used to select subsets of benchmarks.
- `strategy` the strategy used.

- `solved` is `yes` if the benchmark was solved, "`no`" otherwise.
- `time` is the solving time in seconds as a positive float number. Any value that is not a valid floating point number is assumed to be ∞. It is possible to indicate a time even if the benchmark was not solved. Sometimes the solver might give up.

It is simple to add a custom parser. One has to implement a Python class whose constructor takes two arguments: the path to the folder or file containing the data and a list of logics used as filter. This list of logics might be `Null` if the data should not be filtered. If parsing succeeds, the resulting object should have a member `.frame` that contains the parsed data. This member is a Pandas data frame where the rows are the benchmarks and the columns are the strategies. Each cell contains the solving time in seconds stored as a standard float. If the benchmark was not solved by the strategy, the cell stores the float ∞.

Generated schedules are also stored in CSV files. The CSV dialect is as for the input data, but this time only two columns are used: `strategy` and `time`. The file is intended to be read from top to bottom and each line indicates a step in the schedule that executes the strategy `strategy` for `time` seconds. Until schedules are transformed into a script by using `schedgen-finalize.py` the are called *pre-schedules*.

5.4.1 A schedgen Tutorial

This section shows how the schedgen toolbox can be used in practice. We will build and explore an optimal schedule for some hand-crafted, artificial example benchmark data. This data can be found in the file contrib/example_data.csv. The first strategy `base-strategy` solves 20 benchmarks within one second. The five other strategies `extra01` to `extra05` solve up to five benchmarks exclusively such that `extra01` solves five benchmarks not solved by any other strategy, `extra02` solves four, and so on. There is one benchmark `unsolved.smt2` that is not solved by any strategy and a strategy `bad-strategy` that solves only one benchmark in 1.5 seconds. Hence, if the total timeout is six seconds we expect the optimal schedule to run the strategies in the order just presented for one second each. The solving times have been sampled from a normal distribution to ensure the resulting graphs look interesting.

Since the example data is in the CSV format described above the first four lines of `base-strategy` are:

```
benchmark    ; logic ; strategy      ; solved ; time
base01.smt2 ; UF    ; base-strategy ; yes    ; 0.5189
base02.smt2 ; UF    ; base-strategy ; yes    ; 0.2164
base03.smt2 ; UF    ; base-strategy ; yes    ; 0.1754
...
```

The first step is to generate a schedule. To do so we invoke the `schedgen-optimize.py` command:

```
$ schedgen-optimize.py -l UF -s 0.9 1.0 1.1 -t 6 \
    -c -d contrib/example_data.csv contrib/example_schedule.csv
```

The option `-l UF` selects the UF logic, the option `-s 0.9 1.0 1.1` defines the time slices, and `-t 6` sets the total timeout. Hence, every picked strategy can run for either 0.9, 1, or 1.1 second. Overall, the schedule is not allowed to use more than 6 s. Finally, `-c` tells the tool to use the CSV parser, and the option `-d` gives the data location.

The tool then searches the optimal schedule and writes the result to contrib/example_schedule.csv. The first four lines of the schedule are:

```
time  ; strategy
1.100 ; base-strategy
1.000 ; extra01
0.900 ; extra02
...
```

The first column indicates how long the strategy in the second column should be used. As enforced by the synthetic data, the generated schedule uses the strategies in the order outlined above.

Furthermore, the output by the tool predicts how well the schedule will perform. It prints the number of benchmarks solved in the given timeout and the sum of the time needed to solve those benchmarks.

```
['UF']: This schedule solves 35/37 benchmarks. Time needed: 52.19s.
```

Two benchmarks are not solved by this strategy: the benchmark only solved by the `bad` strategy and the benchmark not solved by any strategy. Furthermore, we learn that solving all benchmarks solved by this schedule will take 52.19 seconds. The output also tells us the solving time before order optimization, but in this case nothing changes.

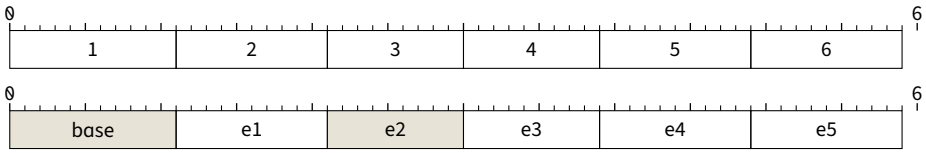


Figure 5.1 The visualization of the example schedule without and with highlights.

The simulator can give us a better prediction of the schedule performance, but let us first visualize the schedule:

```
$ schedgen-visualize.py -t 6 -p out.pgf contrib/example_schedule.csv
```

Figure 5.1 shows the result of this command. The `-t` option works as before and defines the overall timeout. The options `-p out.pgf` tells the visualizer to generate a PGF/Ti*k*Z picture.[30] If this option is omitted, the tool instead opens a window that shows the schedules. Finally, the tool expects a list of pre-schedules to visualize.

Since the different strategies are all used for roughly the same timeout, every strategy is shown as a block of similar size. Because some strategies are used for less than one second, there is a small gap of 0.1 seconds at the end. By default the visualizer simply assigns a number to each strategy when it is used the first time. Strategies often correspond to command line options and those can be long. A number ensures that there are no printing problems, but makes it possible to compare different schedules based on the same strategies.

It is possible to customize the names used for some strategies and to highlight them. This is done with the help of a *shorthand* file: a CSV file with three columns. As in all other CSV files, the fields are separated by a semicolon and there must be a header line. The `strategy` column lists the full strategy string, the `shorthand` column gives the name that should be printed, and the `highlight` can be set to `yes` to indicate that the strategy should be highlighted (`no` otherwise). Strategies not listed in the shorthand files get assigned a number and stay unhighlighted. In Figure 5.1 the base strategy and the second additional strategy are highlighted and every strategy has a custom shorthand name. The command to produce this figure is:

```
$ schedgen-visualize.py -t 6 -c -p out.pgf \
```

[30] If the option `-c` is added, the picture is compatible with ConTEXt.

```
    -a contrib/example_shorthand.csv contrib/example_schedule.csv
```

Let us now simulate the generated schedule. To do so we can use the simulation tool:

```
$ schedgen-simulate.py -l UF -t 6 -c -d contrib/example_data.csv \
    -mu 0.05 -sigma 0.01 -seed 1 \
   contrib/example_schedule.csv simulation_1.txt
```

The options `-l`, `-t`, `-c`, and `-d` are as for the other tools. The tool expects two positional arguments. First, the schedule to simulate and; second, the file to write the results to. The output is always using the GridTPT [30] format, independent of the input format. The remaining options are `--mu`, `--sigma`, and `--seed`. Those control the random jitter that is applied to each solving time in the input dataset. The jitter is sampled from a normal distribution. The options `--mu` and `--sigma` control the mean and the standard deviation, respectively. To ensure simulations can be repeated it is possible to fix an integer seed for the random number generator with the option `--seed`. In this case we choose a slight jitter.

The cumulative density functions generated by the simulation are shown in Figure 5.2. As expected, the graph for the simulation with slight jitter

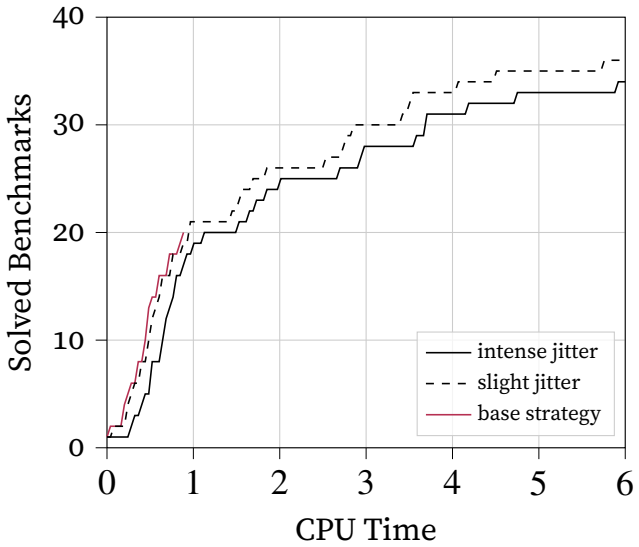


Figure 5.2 Some simulated density functions.

follows almost exactly the base strategy for the first second. The plot also shows the result of using more jitter ($\mu = 0.2$, $\sigma = 0.05$). In this case two benchmarks are no longer solved.[31]

If we want to have a list of the benchmarks not solved by the schedule, we can use the `schedgen-query.py` tool:

```
$ schedgen-query.py -c -d contrib/example_data.csv \
    -q unsolved contrib/example_schedule.csv
special01.smt2
unsolved.smt2
```

The option `-q unsolved` asks the tool to print the list of all benchmarks not solved. An alternative is `-q compare` that shows benchmarks not solved by any strategy. To see the benchmarks that are solved by any strategy together with the minimal solving time, the `-q best` option can be used. Finally, `-q schedule` lists all benchmarks solved by the given schedule together with the predicted solving time (without jitter).

Figure 5.2 is very artificial – it is based on synthetic example data. Section 5.5 and Section 4.5.2 show real examples.

5.5 Evaluation

This section shows some empirical data points related to the real world performance of the schedules generated by schedgen. In Section 4.5 we already saw the schedules used for a practical evaluation. The evaluation of schedgen is simplified by the fact that the generated schedules do not depend on randomized heuristics. Executing the optimizer always results in the same schedule. There are no external factors we must account for.

Nevertheless, it is useful to know how many problems are solved by such schedules. Especially in comparison to simpler approaches, such as using only the best overall strategy, or constructing a schedule with a simple greedy approach. We generate the greedy schedule we start with an empty schedule and then iteratively add the (timeout, strategy) pair that solves the most benchmarks not yet solved. We stop when the timeout is reached.

[31] The script used to generate these graphs is the script contrib/cdf.py in the schedgen repository.

Category	Benchmarks
AUFLIA/20170829-Rodin	3272
AUFLIRA/nasa	18446
AUFLIRA/why	1271
UF/20170428-Barrett	2963
UF/sledgehammer	**4140**
UFLIA/boogie	1191
UFLIA/simplify2	2345
UFLIA/sledgehammer	3462
UFLIA/tokeneer	1872

Table 5.1 Categories with more than 1000 benchmarks. The highlighted category is used by the third scenario.

In machine learning, it is common to use k-fold cross-validation for such evaluations. This means that the data set is split into k subsets of equal size. Then the model is trained on the union of $k - 1$ sets and evaluated of on the remaining set. This is repeated k times and the results are combined (for example by calculating the mean). Each repetition is a *split*. We also use this approach here. and use schedgen to generate one schedule from the training set of each split.

A major problem of such an evaluation is that benchmark libraries, such as the SMT-LIB library, are biased towards the types of problem submitted to the library. Furthermore, the SMT-LIB benchmark collection organizes the benchmarks in categories according to application and submitter. The benchmarks of each category likely share many characteristics. Furthermore, some sets contain thousands of benchmarks, while others contain only tens. To address this issue to some degree, the evaluation only uses categories with more than 1000 benchmarks, and uses three different scenarios. The first is a random sample of 1000 benchmarks from each category. The second is to hold out one category as the test set. This indicates how well a schedule performs on a category that was not seen during the schedule generation. Finally, to see how well such schedule generation can be used to adapt an SMT solver to one application, the last scenario uses only one category of benchmarks.

Solved	Split 1	Split 2	Split 3	Split 4	Split 5	Arith. Mean (σ)
virtual best	1355	1318	1328	1293	1338	1326 (23.1)
generated	**1349**	**1306**	**1317**	**1283**	**1326**	**1316 (24.4)**
greedy	1340	1303	1314	1275	1326	1312 (24.7)
best strategy	1311	1267	1280	1243	1299	1280 (26.7)
PAR-2 score						Arith. Mean (σ)
virtual best	160 501	174 213	170 347	182 938	167 371	171 074 (8316)
generated	**164 388**	**179 811**	**175 453**	**187 851**	**172 102**	**175 921 (8736)**
greedy	169 183	183 040	178 817	192 482	173 655	179 435 (8974)
best strategy	176 844	192 438	187 772	201 248	180 966	187 854 (9606)

Table 5.2 Results of the first scenario.

The experiments use the experimental data gathered for the evaluation in Section 4.5. Instead of running the solver with the schedules, we can estimate the outcome of each schedule with the simulator. All generated schedules use a timeout of 180 s and the time slices 1, 2, 3, 4, 5, 8, 16, 32, 64. Since the benchmark solving rate decreases rapidly with longer timeouts, we can reduce the granularity of the time slices for bigger values. In the benchmarks used in Section 4.5, nine categories contain more than 1000 benchmarks. Table 5.1 lists them. The evaluation uses two metrics: the total number of solved benchmarks and the PAR-2 score. The PAR-2 score is the sum of all solving times, plus twice the timeout for each benchmark not solved. Hence, a lower PAR-2 score is better. The PAR-2 score is used for scoring at competitions such as the SAT competition [59].

Overall, the first scenario uses 9000 benchmarks and 5-fold cross-validation over all benchmarks. Therefore, the schedules are generated on 7200 benchmarks and evaluated on 1800 benchmarks. Table 5.2 shows the result of the evaluation. The first part shows the total number of benchmarks solved by the virtual best solver, the generated optimal schedule, the schedule generated by the greedy approach, and the single strategy that solves the most benchmarks. Here, the virtual best solver is constructed from all available strategies. The second part shows the corresponding PAR-2 scores. We see that the generated optimal schedule is better in all splits except for the fifth split. In this case, both the greedy and the generated schedule solve the same number of problems. Nevertheless, even in this case the generated

Solved	Split 1	Split 2	Split 3	Split 4	Split 5	Arith. Mean (σ)
virtual best	248	268	265	271	271	265 (10)
generated	**235**	**262**	**261**	**264**	**265**	**257 (13)**
greedy	237	255	259	261	262	255 (10)
best strategy	222	245	244	252	245	242 (11)

PAR-2 score						Arith. Mean (σ)
virtual best	209 094	202 099	202 821	201 265	200 737	203 203 (3388)
generated	**214 328**	**204 673**	**204 829**	**203 835**	**203 863**	**206 306 (4508)**
greedy	215 158	207 890	206 677	205 539	206 027	208 258 (3957)
best strategy	218 768	210 590	210 727	208 061	210 516	211 732 (4086)

Table 5.3 Results of generating schedules for UF/sledgehammer.

schedule has a better PAR-2 score. Overall, the PAR-2 scores also show that the generated optimal schedule is better than the greedy schedule and much better than not using a schedule at all. Furthermore, the variance is also reduced by using an optimal schedule. While the increase in solved benchmarks seems small, the window for improvement is only a small percentage of the total number of benchmarks: on average the best strategy alone already solves 1280 benchmarks, and even the virtual best solver solves only 14 benchmarks more then the greedy schedule.

The second scenario uses the same 9000 benchmarks, but for each split a different category is used as a test set. Since there are nine categories, this scenario is a 9-fold cross-validation with training sets containing 8000 benchmarks, and test sets containing 1000. Note that the share of benchmarks that are unsolvable for veriT (e.g., satisfiable benchmarks) can vary widely from category to category. As a consequence, the mean of the results is not very meaningful for this experiment.

The results are in Table 5.4 on page 138. To save space, the names of the categories are simplified. The category “Lsledgehammer” corresponds to the category UFLIA/sledgehammer. This scenario is, in a certain sense, a stress test for the schedules. Each category might have some characteristics not shared with any other category. The optimal schedule seems to struggle with the AUFLIA/20170829-Rodin category. It only improves slightly over the greedy schedule, while the virtual best solver is significantly better. For three categories both schedules and the virtual best solver solve the same number

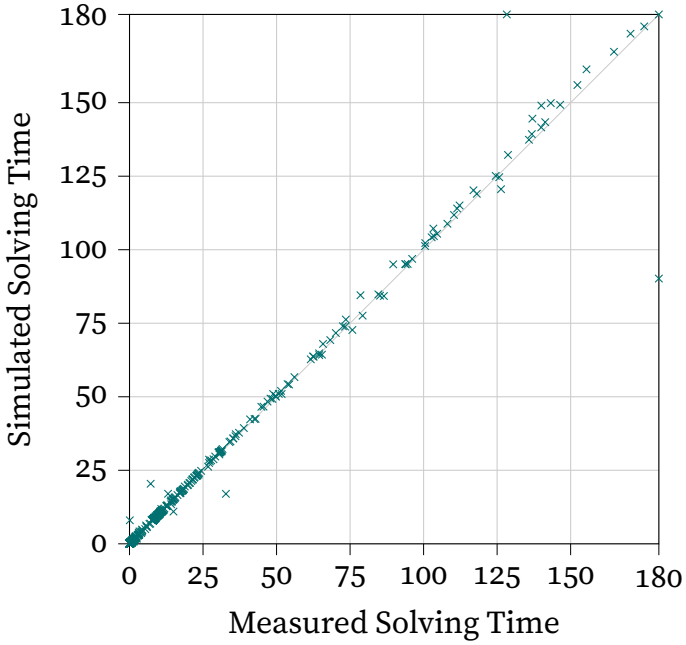


Figure 5.3 Simulation versus reality.

of benchmarks. In one case (UFLIA/tokeneer) the optimal schedule has the worst PAR-2 score. Furthermore, in two cases (AUFLIA/20170829-Rodin and UFLIA/simplify2) the heuristic optimization of the order slightly increased the PAR-2 score. Overall, optimal schedules are clearly doing well in this scenario too.

The third and final scenario uses only the category UF/sledgehammer. This category contains problems generated by Sledgehammer [23]. Since veriT is also used by Sledgehammer, this is a sensible choice. The UF/sledgehammer category contains 4140 benchmarks. Again, 5-fold cross validation is used: 3312 benchmarks for optimization and 828 for testing. Table 5.3 shows the results that mirror the other two scenarios. For two splits, all schedules solved the same number of benchmarks, and for the optimal schedule improves only slightly in the first split. Both effects are probably the result of the relative small number of benchmarks in the test set. In any case, the optimal schedule improves on the PAR-2 score over the greedy schedule.

We also collected the predicted sum of solving times for the schedule before and after order optimization. The order before optimization is an implementation detail of the toolbox and not deterministic. The average ratio between the time after optimization and before is 0.71.

To finish this section, let us have a look at the real world behavior of schedules in comparison with the simulation. This allows us to judge how reliable the simulated results presented so far are. The data gathered for Table 4.1 in Section 4.5.1 serves as the base for this. Figure 5.3 shows a scatter plot of the solving times for the 180 s schedule with quantifier simplification by unification. The closer a point is to the diagonal, the more accurate the simulation was. This shows that the simulation is fairly accurate. In a few cases, benchmarks predicted to be solved are not solved in reality and vice versa. Those are benchmarks that are solved just before the time slice of a strategy ends. In this case, a delay can make the benchmark unsolvable. The opposite is also possible: if a benchmark is solved during data gathering after the time slice ends, the jitter can reduce the solving time below the timeout. Overall, this graph indicates that the simulation is a good tool to assess the efficacy of a schedule.

The evaluations have been performed on a laptop with an Intel i7-7600U CPU and 16 GB of memory. Generating the 19 schedules took 39 min 16 s overall. Therefore, generation time is short. To generate the schedules for veriT at the SMT competition, GNU parallel [94] was used to generate the schedules for the different logics in parallel.

Solved	Rodin	nasa	why	Barrett	sledge-hammer	boogie	Lsledge-simplify2	hammer	tokeneer	Arith. Mean (σ)
virtual best	764	984	982	535	337	689	995	424	922	720 (255.8)
generated	**746**	**984**	**982**	**527**	**332**	**677**	**992**	**406**	**922**	**714 (260.1)**
greedy	745	984	982	517	328	673	991	402	922	711 (262.4)
best strategy	716	981	979	491	296	648	893	374	922	693 (264.4)
PAR-2 score										Arith. Mean (σ)
virtual best	85 637	5761	6483	167 838	238 974	112 173	1994	208 430	28 080	100 939 (92 338)
generated	**92 244**	**5801**	**6591**	**172 126**	**242 243**	**118 134**	**3532**	**215 400**	**28 085**	**103 801 (94 333)**
greedy	95 380	6146	6697	176 215	245 105	120 403	9799	218 310	28 081	105 909 (94 946)
best strategy	102 387	6840	7585	183 730	253 850	127 337	38 707	226 300	28 081	110 879 (95 467)

Table 5.4 Results of the second scenario.

5.6 Related Work and Outlook

As we saw, automatically generated schedules can solve the problem of adapting theorem provers to the limited resources of the real world. Because of this, they are widely used by many systems. Nevertheless, instead of executing a fixed schedule, one could also build a system that picks a strategy based on some characteristics of the problem. In general, this approach is called *algorithm selection* [83]. The name is inspired by the fact that algorithm selection can be used in many domains, some of which use clearly distinct algorithms. In the case of theorem proving, the strategies take the role of the algorithms to select. Hence, we will use the term *strategy selection*. A third aspect of working with strategies is the library of used strategies. schedgen uses a predefined set of strategies. While this set is usually defined by people familiar with the theorem prover, one could also think about an automatic system that generates strategies. This section is about systems that address one or multiple of those three aspects. As we will see, they are often tightly coupled. The goal of this section is not to provide a comprehensive history of this field, but to highlight a few interesting systems.

The theorem prover Gandalf is considered the pioneer of strategy scheduling for theorem provers [67,100]. It participated in the first CASC competition at the CADE conference in 1996. In this competition, however, it used an ad hoc form of *strategy selection* which is also described in the Gandalf system description [92]. Essentially, Gandalf picks one of multiple resolution based calculi depending on the properties of the input problem. The performance of this approach at the competition was not satisfactory. The author of Gandalf, Tanel Tammet, says about this via email:

> "The experiments I did after the first CASC indicated that (a) it is pretty hopeless to find a really good strategy for a problem by simple means (b) most importantly, giving more time to prover leads to a logarithmic performance growth, i.e. given like 10 times more seconds would only make it very slightly better.
>
> Hence the choice to use available time more sensibly by trying out a number of strategies with strongly limited time: batches of strategies with iteratively increasing time limits for the repetitions of the batch."

This new approach of time slicing was used at the CASC-14 competition the following year[32] where Gandalf won the MIX category. Nevertheless, the strategy schedule used in this version is also not entirely fixed. Instead, Gandalf picks one of multiple schedules. A fixed part of the schedules were constructed manually through extensive experiments, but the prover can also adapt them based on some parameters, such as the maximal term depth. Hence, Gandalf already combined strategy scheduling with strategy selection to a limited degree.

Directly inspired by Gandalf is the schedule generator p-SETHEO [100] published in 1998, one year after Gandalf won the CASC competition with the help of strategy scheduling. The authors of p-SETHEO remark that finding an optimal schedule is strongly NP-complete. For this reason, they use a heuristic procedure [100]. The procedure works with a fixed list of strategies and a mapping from strategies to timeouts. Furthermore, the procedure maintains a factor ΔT which is set to a fixed fraction of the total timeout. At start, every strategy mapped to a zero timeout. Then the mapping is modified by adding a time ΔT to one strategy such that the number of solved benchmarks is maximally increased. If no such strategy exists, ΔT is increased by the fixed fraction of the timeout. Once the sum of mapped timeouts reaches the total allowed timeout, the procedure terminates and the mapping represents the generated schedule. The p-SETHEO tool itself can execute strategies in parallel on multiple processors, but the authors do not address possible memory contention.

The perhaps best known user of strategy scheduling is the Vampire theorem prover. It used strategy schedules to win numerous competitions. The schedules are called *portfolio modes* and can be selected by application (e.g., CASC or SMT-COMP). Since the portfolio mode is one of the main reasons Vampire performs so well at competitions, the method of how schedules are generated was historically shrouded in mystery and has never been fully published. Nevertheless, we can learn a bit about the portfolio modes of Vampire from the source code and from information provided by the Vampire developer Giles Reger.[33] In general, Vampire sorts input problems into buckets based on certain properties, such as the number of clauses. This is achieved

[32] There was no system description for Gandalf at CASC-14, but the system description for CASC-15 [93] outlines the strategy schedule approach.

[33] Vampire is available at https://github.com/vprover/vampire. The source code for the portfolio modes is in the CASC folder.

by simple `if` statements that choose the hard coded schedule assigned to a bucket. The selected schedule is then handed to a schedule executor. The executor can distribute strategies to multiple cores and after a schedule is finished, it runs the schedule again with all timeouts doubled. The schedules themselves are generated using a greedy algorithm. Some Git commits indicate that sometimes schedules are adjusted manually. There are commits that change only a couple strategies. Furthermore, code comments such as "Empirically sorted (order somewhat guessed)" hint at a creation process that is partially manual.

Another aspect of the Vampire portfolios is the search for strategies. This happens in two steps. First, a large database is built by running random strategies on random problems. Second, the database is refined by local search around successful strategies. This seems to be similar to the process used by systems such as MaLeS described below. A special element of this process is that those two steps are executed concurrently all year round.

In addition to the extensive use of schedules, Vampire uses another technique to adapt to limited resources. This is the aptly named *limited resource strategy* [82]. The limited resource strategy adapts the search to the time limit. Since Vampire is a saturation based solver, the proof search is implemented as a saturation loop. This loop maintains two sets: *active* and *passive*. All initial clauses are added to the passive set. Every iteration moves one clause from the *passive* set to the *active* set and also performs all inferences between this clause and the clauses in the *active* set. The results of these inferences are added to the *passive* set. The decision which clause to pick is based on a heuristically calculated weight. Usually, clauses in the *passive* set are stored in a priority queue according to weight. The loop ends when either the empty clause is inferred, or the *passive* set is empty. Since the number of clauses grows rabidly, there are clauses in the passive set that will never be picked. The key idea of the limited resource strategy is to estimate which clauses in the *passive* set will not be reached before timeout and to remove these from the *passive* set. Since this set is maintained as a priority queue, this can be achieved by estimating a cutoff weight. Note, that as the *active* set grows, every loop uses more time. Hence, the estimate must be adapted as saturation progresses. If the estimate is perfect, then every problem that is solved without the limited resource strategy within a time t can still be solved if the limited resource strategy is used with the timeout t. Since the estimate is not perfect, some problems are lost in practice. Nevertheless, there is a secondary effect. Since a short timeout forces the limited resource

strategy to remove more clause, problems might be solved faster if a shorter timeout is used.

Unfortunately, it is not clear how something like the limited resource strategy can be implemented within an SMT solver. The strategy relies on the fact that a saturation loop performs inferences in a monotone fashion by using a queue. Within an SMT solver, however, the ground models changes unpredictably. Hence, for example, the instantiation module cannot reliably predict which instances will not be used before the timeout.

The MaLeS system [67] covers all aspects of working with strategies and has been used with the theorem provers E, LEO-II, and Satalax. It uses a machine learning model to select strategies from a list of admissible strategies. An interesting aspect of MaLeS is that instead of selecting only one strategy it uses the model to dynamically build a schedule. To pick admissible strategies it uses local search. To select strategies MaLeS uses a function that predicts the runtime of the solver for a specific strategy on an input problem. To do so, MaLeS extracts features, such as the number of literals, from the input problems. Features are represented as real numbers. The prediction function maps this feature vector to a real number: the predicted runtime. The prediction function is built by regression using a Gaussian kernel.

The prediction function is used by MaLeS to dynamically build a schedule. Every schedule starts by a short list of fixed strategies. This list is built in a greedy fashion. It contains the strategies that solve the most benchmarks. After this initial list is exhausted, the prediction function is used to find the strategy with the shortest predicted runtime. This strategy is used subsequently with the predicted runtime as timeout. In any case, if the strategy fails, the prediction function is updated with this data point. This process is repeated until the problem is solved, or the overall timeout is reached. Note, that it is possible that the same strategy might be tried multiple times with increasing timeout.

MaLeS does not use machine learning to generate the set of admissible strategies. Instead, it relies on local search. It starts with a queue filled with strategies defined by the user. Then a loop is executed that starts with popping one strategy of the queue. The process ends when the queue is empty. On every round the current strategy is tested on every training problem. If it is able to solve the problem faster than any other strategy tested so far, it is modified randomly. These new random strategies are then also tested on the problem, and the overall fastest strategy is stored and pushed onto the queue. The resulting set of admissible strategies are the strategies that solved any

problem faster than any other strategy. This approach is a specific instantiation of the ParamILS algorithm configuration framework [61]. Another system in the same school is the BliStr tool [95]. This system is specialized to generate strategies for the E theorem prover. Instead of evaluating strategies on one problem at a time, it interleaves evaluation on a small sets of problems with short timeouts with evaluation on all benchmarks with long timeouts.

The Sledgehammer tool also has to content with managing limited resources. This is deeply ingrained into the tool. During both the proof search and the preplay phase Sledgehammer must execute multiple tools within the time limits. Since it relies on external tools to do the actual solving, this is a fundamental aspect of Sledgehammer. Isabelle/HOL itself already provides infrastructure to distribute processes to multiple processors as appropriate [98]. Hence, Sledgehammer just has to provide a list of either parameterized solvers in the case of proof search, or of tactics in the case of preplay. The list of tactics for preplay is handcrafted and starts with simple, fully automatic tactics, and then uses more complex tactics such as `smt`. The of sequence of tools to use during the initial fact-filtering phase, on the other hand, was generated automatically from a large scale experiment in a greedy fashion [39]. The list does not only store solver and how they are parameterized, but also stores parameters for Sledgehammer itself. This includes the number of background lemmas to export and what encoding should be used.

So far, SMT solvers have only worked with strategies in a very limited fashion. A testament to this is the Par4 [97] system. It is a small script that can execute multiple solvers in parallel on an input problem. It won multiple categories at the SMT-COMP 2019. In subsequent years, the rules of the competition were changed, and such wrappers can no longer compete in the main competition. The solvers CVC4 and cvc5 [11] use a simple hand-crafted schedule script at the SMT-COMP. The script used by veriT is based on this script.

A special case is the Z3 SMT solver. It supports a special language that can be used to control how the solver tackles a problem. The indent is that users can adapt Z3 to their problem domain by writing scripts that control the solving process [48]. This scripting language was added to Z3 to allow users to adapt the internal working of the solver in a fine-grained manner. Hence, the language replaces the simple control by using command line parameters. Since it also supports scripting, schedules can directly be encoded in this

language. There is a tool based on machine learning that can synthesize such scripts [8]. It goes beyond generating pure schedules by synthesizing valid expressions in the language.

Finally, MachSMT is a tool that uses machine learning for strategy selection [89]. It is inspirited by the SATzilla tool [101] for SAT solvers. To select a strategy (or solver) it can use two types of models. The first one is an *empirical hardness model*. Such a model is a function that predicts the runtime of any solver on a given input problem. The model used by MaLeS is an empirical hardness model. MachSMT uses such a model by selecting the strategy with the lowest predicted runtime. The second type of model is a *pairwise ranking comparator* that predicts the faster strategy of a pair of strategies. Such models must be trained for each pair of strategies. MachSMT selects the strategy such that the number of other strategies that are predicted to be faster is minimized. MachSMT uses boosted decision trees as models. Typical for a machine learning based approach, MachSMT extracts a feature vector from the input problem. Some features are specific to the SMT theory, such as the number of occurrences of theory symbols. Since the number of features is large, MachSMT uses principal component analysis to reduce the dimension of the feature vector.

The schedgen toolbox would profit a lot from a strategy finding module. This module should also produce the experiential data for schedule generation in addition to the generate strategy. For example, the user could start this module with a fixed timeout. The module then produces as many good strategies as it could find within the timeout. The user could then go on to generate a schedule without any additional work. Furthermore, it would be great if schedgen provided a schedule execution script that can execute strategies in parallel. One challenge with this is the limited system memory. Especially in proof production mode, veriT can use high amounts of memory. Multiple instances of veriT running in parallel would compete among each other for the available memory. In the worst case, a solver process could be killed prematurely, and the schedule is not executed as expected. One solution is to have one privileged process that can us as much memory as it wants, while the other processes compete for the remaining memory. The privileged process executes the schedule like usual, while the unprivileged processes pick (timeout, strategy) pairs later in the schedule. If an unprivileged process executes a strategy until timeout, that pair is removed from the schedule. Therefore, in the worst case, the schedule is executed in the usual

sequential manner by the privileged process. Such a setup can be achieved, for example, with Linux cgroups[34].

While strategy selection, as performed by tools like MachSMT, is very useful, it would probably best be explored as a tool outside schedgen. Currently, all tools in schedgen are structure around the shared concept of a fixed schedule and a strategy selection model would not fit very well into this architecture.

> Es kommt die Zeit, deine Zeit,
> zeig dich jetzt, sei bereit.
> Steh auf, steh auf – lauf.
> Lauf, schau nach vorn, kein zurück,
> bau dich auf, Stück für Stück
> Steh auf, steh auf – lauf.
> Lauf, spreng die Norm, deine Form,
> stell dich jetzt, komm nach vorn.
> Mit dem Herz in der Hand,
> mit dem Kopf durch die Wand.
>
> —Klangstabil, Lauf, Lauf!

[34] See https://www.kernel.org/doc/Documentation/cgroup-v2.txt

VI Conclusion and Outlook

The previous three chapters showed three contributions that strengthen the potential of SMT solvers as backends for proof assistants. Each contribution aims at a different element of the SMT solver.

Chapter III addressed the generation of proofs that can be consumed by the proof assistant. The improvements made to the Alethe proof format were not only informed by theoretical and implementation considerations, but also addressed shortcomings discovered during the implementation of a proof reconstruction procedure. This was a successful approach. The reconstruction is now mature, and the veriT powered `smt` tactic is regularly proposed by Sledgehammer. On 7 July 2022 the Archive of Formal Proofs contains 611 calls to the veriT powered `smt` tactic.[35] The interoperability between systems will be a major factor in the development of SMT proof formats. A good format should not only simplify the interaction between one solver and one consumer, but should also be easy to support by multiple solvers. This can be simplified with the help of intermediary tools that can elaborate coarse-grained proofs and eliminate solver-specific extensions. The outlook for the Alethe format discussed in Section 3.3.3 proposes such features.

Chapter IV discusses a novel quantifier simplification method for SMT solvers. Since proof assistants use expressive logics, good support for quantifiers is necessary for SMT solvers that should serve as backends. The simplification method uses unit clauses that have a quantified formula as their literal to simplify nested quantifiers. This structure appears in problems generated by proof assistants. For example, the unit clause can be a user defined lemma that is used as guard in another lemma. The empirical evaluation also showed that when the simplification method is applicable, the remaining problem is often solved very fast. This is a good property for a backend solver.

Quantifier simplification by unification was motivated by the lack of normalization of quantified formulas. Quantified formulas remain nested and hard to instantiate. However, there is no fundamental reason why SMT solvers do not fully clausify and skolemize the quantified part of the input problem. This normalization is standard in resolution-based theorem

[35] The archive is available at https://www.isa-afp.org/.

provers. Implementing this in an SMT solver would require some adaption of the instantiation techniques. For example, trigger inference must be performed before the normalization, and the search for conflicting instances must take the function symbols introduced by a Tseitin encoding into account. In practice, this means that the instantiation module must be reimplemented. On the one hand, normalizing the entire problem up front would simplify the overall solver. We no longer have to reason with arbitrarily complex quantified formulas during the solving process. On the other hand, the normalization destroys the structure of the problem. This trade-off should be studied for SMT solvers in a rigorous manner. Currently, the lack of normalization in SMT solver has historical reasons and is not backed by empirical evidence.

Chapter V presented a feature-rich toolbox to work with strategies and strategy schedules. The toolbox cannot only be used to generate strong strategy schedules, but can also be used as a strategy search tool. For example, we used it to pick a fixed set of strategies for the Isabelle/HOL integration. In general, it is dangerous to use strategy schedules when a solver is used as a backend, like in the `smt` tactic. Since strategies are scheduled by elapsed time, they can aggravate the nondeterminism of automated theorem provers. Even a solver, such as veriT, that is perfectly deterministic with respect to the inferences performed can become nondeterministic if a schedule interrupts the critical strategy prematurely on a slower computer. The solution used by Sledgehammer is to perform the scheduling on the Isabelle/HOL side and to store the successful strategy by letting the user insert it into their proof script. This might not be feasible for all applications. Another approach would be to use a measure other than elapsed time in the strategy schedule. Theoretically, this measure is only reacquired to grow monotonically, but a roughly linear growth rate would make it more useful for practical application. Some SAT solvers use such measures to switch between modes during solving [21].

All three chapters show that SMT solvers can be adapted to specific applications. Either by extending them, or by providing new tools and interfaces. SMT solvers are not fixed black boxes that consume logical problems and spit out answers. They are complex software systems that can be adapted. Unfortunately, this requires deep knowledge of the solver. Furthermore, despite the flexibility of the CDCL(T) calculus and theory combination, users of SMT solvers seldom develop new theories and theory solvers independently. This gap between the solver and its users was already identified and characterized

as the "Strategy Challenge in SMT Solving" [48]. The SMT community should take this challenge to strengthen SMT solving.

One way to narrow the gap between the SMT solver and a specific user – proof assistants – is to move to higher-order logic on the SMT solver side. The solver can then use an encoding optimized for its solving techniques, or even use solving techniques adapted to higher-order logic throughout. To move superposition-based theorem provers and SMT solvers to higher-order logic was the main direction of the Matryoshka project [25]. The approach taken by the Matryoshka project to achieve this was a stratified architecture: the first-order reasoning stays intact, but is extended with rules to reason with higher-order constructs. On the SMT side this can be achieved by extending the instantiation procedures [16].

We can draw inspiration from this approach to build architectures that provide more control to the users. The underlying CDCL(T)-based solver architecture remains mostly unchanged, but is carefully extended with user controlled inferences. Some SMT solvers expose interfaces that allow users to manipulate the internals of the solver. The solver Z3, for examples, features a strategy-based configuration language [48]. This approach, however, exposes the control flow of the solver to the user and is not stratified.

Despite modern solvers lacking support to extend them conservatively, users use an existing feature to extend solvers "in-officially". This feature is trigger-based instantiation. By carefully choosing triggers to achieve certain effects, users can control the reasoning process. Triggers are, for example, extensively use by the F* system [3] and by TLAPS's SMT backend [36]. To see how triggers can be used in this way, consider a problem that encodes lists. The problem could contain the lemma $(\forall x\ xs.\ (head\ (cons\ x\ xs)) \approx x)$. To ensure the *head* function is applied during quantifier instantiation, the user will add the trigger $(head\ (cons\ x\ xs))$. In fact, the user might know that it is only sensible to instantiate this lemma when the trigger matches. Currently this cannot be enforced without turning off all other instantiation methods. This approach can be used to implement support for entire theories [43].

A carefully designed extension of triggers can provide more control to the user. Such a speculative extension could be based upon the well researched field of term rewriting. Instead of triggers, the user provides rewrite rules together with a description of how and when the rule should be applied. In the example above, the user would add the rewrite rule $(head\ (cons\ x\ xs)) \mapsto x$. Even the developer of the solver can profit from such an extension. Since it is easy to provide proofs for the application of rewrite rules, the extension can

be used to implement proof-producing preprocessing and theory reasoning for free.

Rewriting as a mechanism to define theories has been studied as part of the *deduction modulo theory* framework [41]. In this approach, reduction rules take the role of axioms when defining theories for the purpose of automated reasoning. Beyond proof theoretic studies, this approach is also implemented in some practical automated reasoning tools [32]. The use of rewriting to extend SMT solves has also been considered. For example, in the CVC4 system as a way to simplify the implementation of theory solvers for complex theories such as non-linear arithmetic [81].

Overall, higher-order logic will provide the user with the expressiveness to encode their problems directly. The extension mechanism based on rewriting and built in theories will provide the user with a mechanism to express their domain knowledge and to ensure the solver can solve their problems.

Résumé

Cette thèse présente trois contributions qui ont pour objectif d'améliorer l'utilité des solveurs SMT comme backends pour les assistants de preuve. Les solveurs SMT sont un type de prouveurs de théorèmes entièrement automatisés. Ils intègrent le raisonnement propositionnel avec des théories et sont excellents dans la résolution de problèmes utilisant peu de quantificateurs. Dans cette thèse, nous utilisons le solveur SMT veriT [31]. Les assistants de preuve, également appelés prouveurs interactifs, sont des outils qui permettent aux utilisateurs d'écrire des preuves formellement vérifiées. Ils fournissent des mécanismes sophistiqués pour travailler avec des théories mathématiques étendues, ou des problèmes de vérification compliqués.

Les assistants de preuve doivent faciliter la vie de l'utilisateur. L'utilisateur doit pouvoir se concentrer sur les parties délicates de la démonstration et ne pas être distrait par les aspects techniques de l'assistant. L'automatisation est un moyen d'y parvenir, et peut être implémentée entièrement dans l'assistant de preuve, ou utiliser des outils externes, tels que les solveurs SMT. Lorsque l'automatisation fonctionne bien, les obligations de preuve qui sont « triviales » peuvent être résolues automatiquement. Un exemple concret d'une telle intégration est l'outil Sledgehammer [24] qui fait partie de l'assistant de preuve Isabelle/HOL [22].

La figure 1 montre la chaîne de traitement de Sledgehammer. Ce dernier est invoqué par l'utilisateur à l'aide de l'interface Isabelle/HOL. Lors de son invocation, il effectue deux étapes : le filtrage des faits et le *preplay*. Pour commencer le filtrage des faits, Sledgehammer encode l'obligation de preuve courante et les faits sélectionnés dans les théories de base de la logique du prouveur utilisé. Ensuite, il invoque des prouveurs automatisés de théorèmes externes. Cette étape réussit si au moins un d'entre eux obtient une preuve. Cependant, Sledgehammer ne peut pas utiliser cette preuve directement. En effet, Isabelle/HOL (ainsi que de nombreux autres assistants de preuve) n'acceptent que les preuves exprimées dans leur propre formalisme.

Pour trouver une telle preuve, Sledgehammer effectue la deuxième étape : le preplay. Il invoque à nouveau des outils automatisés selon un schedule. Cependant, les outils invoqués sont cette fois-ci internes à Isabelle/HOL et utilisent donc son formalisme internes. Pour aider les outils internes à résoudre l'obligation de preuve, Sledgehammer utilise le noyau insatisfiable extrait de la première étape. Il ne fournit que les faits du noyau aux outils internes.

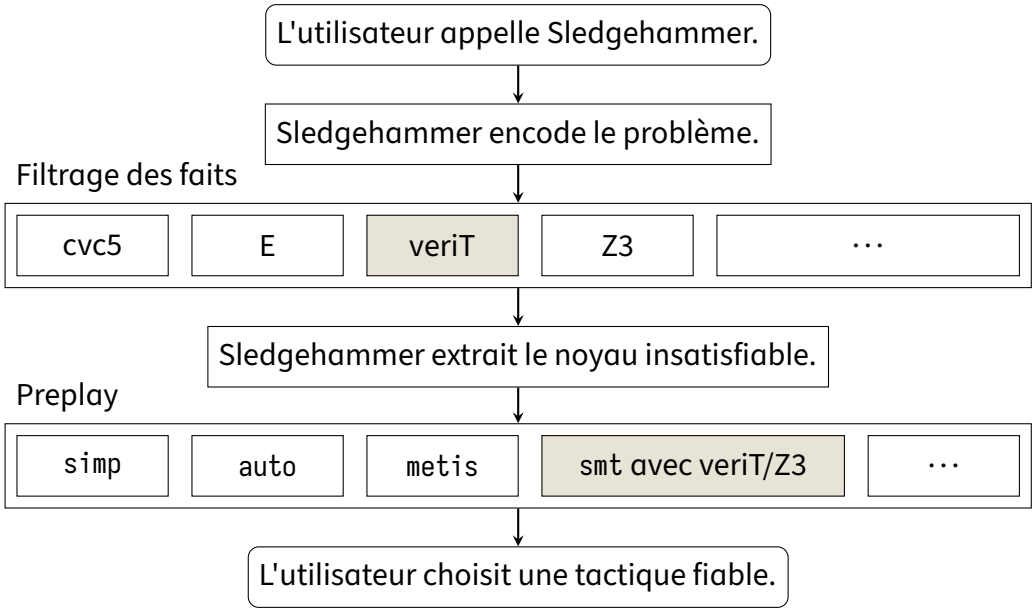


Figure 1 L'interaction entre l'utilisateur, les prouveurs automatisés de théorèmes, et Sledgehammer.

Par conséquent, les théorèmes automatisés agissent comme des filtres de faits dans la première étape. Si un outil interne obtient un résultat pendant le preplay, il est proposé à l'utilisateur, qui peut choisir l'incorporer dans sa preuve. La tactique `smt` est un cas particulier d'outil interne : elle appelle aussi un prouveur automatisé de théorèmes externe, mais récupère une preuve complète et reconstruit la preuve dans le noyau de confiance. L'utilisateur est d'autant plus satisfait que le taux de réussite de Sledgehammer est élevé.

Un cas d'échec particulièrement frustrant est celui du preplay. En effet, l'utilisateur sait qu'il existe une preuve, mais ne peut pas l'utiliser. Une façon de réduire le taux de ce type d'échec est d'étendre la tactique `smt` pour rejouer différentes preuves. La complexité d'implémentation d'une telle procédure et sa fiabilité dépendent en grande partie de la qualité de la preuve générée par le prouveur automatisé de théorèmes. Nous abordons ce sujet du point de vue du solveur SMT. Notre format de preuve Alethe améliore et unifie les travaux antérieurs sur la génération de preuves à partir de solveurs SMT. La grande majorité de ces améliorations a été inspirée par l'expérience acquise lors d'un travail substantiel visant à la reconstruction de preuves Alethe dans l'assistant de preuve Isabelle/HOL.

Pour permettre à l'utilisateur d'exprimer des théorèmes mathématiques complexes, les assistants de preuve utilisent des logiques expressives. Dans le cas de Isabelle/HOL, il s'agit de la logique d'ordre supérieur. En revanche, les solveurs SMT raisonnent principalement sur des problèmes du premier ordre sans quantificateur. Pour combler ce fossé, l'approche principale consiste actuellement à utiliser un encodage de l'obligation de preuve en logique du premier ordre et un processus d'instanciation de quantificateurs du côté du solveur SMT. La deuxième contribution présentée améliore cette chaîne de traitement. Elle ajoute une étape de prétraitement au solveur SMT qui peut éliminer les quantificateurs imbriqués en utilisant l'unification. En pratique, les prouveurs automatisés de théorèmes doivent utiliser des heuristiques pour résoudre de nombreux problèmes. Les solveurs SMT ne diffèrent pas des autres systèmes à cet égard. Typiquement, les heuristiques peuvent être paramétrées. Un paramétrage spécifique d'un solveur est appelé une stratégie, et la meilleure stratégie diffère généralement d'un problème à l'autre. Par conséquent, l'utilisateur d'un solveur SMT peut tirer profit de l'utilisation d'une stratégie appropriée et de l'essai de plusieurs stratégies. La troisième contribution est une panoplie d'outils destinés à faciliter l'utilisation et le choix de ces stratégies. Un des outils majeurs propose d'utiliser l'optimisation linéaire en nombres entiers pour générer des schedules de stratégie. D'autres outils concernent la simulation et l'analyse des schedules. Cet ensemble d'outils est utilisé pour déterminer les stratégies permettant de résoudre certains problèmes engendrés par Isabelle/HOL.

Dans l'ensemble, cette thèse montre que lorsqu'un solveur SMT est traité comme un système logiciel complexe au lieu d'une boîte noire, de nombreuses voies d'amélioration s'ouvrent.

Améliorer les preuves. Le format de preuve utilisé par le solveur SMT veriT s'appelle « Alethe » et s'appuie sur les formats utilisés précédemment par veriT. Ces formats remontent à dix ans [20]. Depuis sa création, ce format a été affiné et étendu [12]. Le travail d'implémentation de la reconstruction de preuve pour la tactique `smt` a mis en évidence les lacunes des preuves générées par veriT. Il a également fourni l'occasion de peaufiner le format de preuve. De plus, il n'y avait pas de documentation complète ni de spécification des règles de preuve disponibles. Cette thèse comble cette lacune : il s'agit d'une documentation complète du format de preuve Alethe.

Outre la documentation du format de preuve, mes contributions aux preuves SMT comprennent également de nouvelles règles de preuve ainsi

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 vr4)))
(step  t9.t1 (cl (= z2 vr4)) :rule refl)
(step  t9.t2 (cl (= (p z2) (p vr4)))
          :rule cong :premises (t9.t1))
(step  t9 (cl (= (forall ((z2 U)) (p z2)) (forall ((vr4 U)) (p vr4))))
          :rule bind)
...
(step t14 (cl (forall ((vr5 U)) (p vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((vr5 U)) (p vr5))) (p a)))
          :rule forall_inst :args ((:= vr5 a)))
(step t16 (cl (not (forall ((vr5 U)) (p vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Figure 2 Exemple de preuve retourné.

que certaines améliorations, et la syntaxe basée sur SMT-LIB qui utilise une séquence de commandes au lieu d'une structure imbriquée.

Alethe suit quelques principes de conception fondamentaux. Tout d'abord, les preuves d'Alethe doivent être faciles à comprendre par les humains, afin que travailler avec elles soit facile. Deuxièmement, le langage du format doit correspondre directement au langage utilisé par le solveur. Comme de nombreux solveurs utilisent le langage SMT-LIB, Alethe utilise également ce langage. Par conséquent, la logique de base d'Alethe est la logique du premier ordre multi-sortée de SMT-LIB. Troisièmement, le format doit être uniforme pour toutes les théories utilisées par les solveurs SMT. A l'exception des clauses pour le raisonnement propositionnel, il n'y a pas de syntaxe dédiée à une théorie en particulier.

La Figure 2 montre un exemple de preuve Alethe engendrée par veriT. Elle introduit d'abord deux hypothèses (`h1` et `h2`). Ensuite, elle utilise une sous-preuve pour renommer les variables liées. La commande `anchor` lance la sous-preuve et introduit le contexte $z2 \mapsto vr4$. Alethe utilise des contextes pour exprimer de manière précise des étapes de raisonnement sur les variables et

leurs associations. La sous-preuve comprend les étapes `t9.t1` et`t9.t2`. Tandis que `t9` conclue la sous-preuve. Enfin, la preuve se termine par les étapes d'instanciation et de résolution (`t14` à `t17`).

Les assistants de preuve, tels que Isabelle/HOL, n'acceptent pas les preuves qui ne passent pas par le noyau d'inférence de l'assistant. Par conséquent, la tactique `smt` de Isabelle/HOL exécute un solveur SMT, analyse la preuve puis la *reconstruit* à travers le noyau. Cette reconstruction permet l'utilisation de prouveurs externes. La tactique `smt` a été développée à l'origine pour le solveur SMT Z3 [28,47]. Elle utilise quatre étapes pour travailler avec les preuves d'Alethe.

1. Elle nie le but de la preuve (pour avoir une preuve par réfutation) et encode la résultat en logique du premier ordre. L'encodage élimine les λ-abstractions. Pour ce faire, il remplace chaque λ-abstraction par une nouvelle fonction et crée des opérateurs « app » correspondant à l'application de fonctions binaires du premier ordre.

2. Elle analyse la preuve trouvée par veriT (s'il y en a une) et l'encode comme un graphe acyclique dirigé, avec ⊥ comme seule conclusion. Ce graphe peut contenir des sous-preuves avec des hypothèses locales. La structure de la preuve est légèrement modifiée pour simplifier la reconstruction.

3. Elle convertit les termes SMT-LIB en termes typés Isabelle/HOL et inverse également l'encodage utilisé pour convertir les termes d'ordre supérieur en termes de premier ordre.

4. Elle parcourt le graphe de preuve, vérifie que toutes les assertions d'entrée correspondent à leur contrepartie Isabelle/HOL et reconstruit ensuite la preuve étape par étape en utilisant les primitives du noyau.

Pour améliorer la reconstruction des preuves dans la tactique `smt` nous avons amélioré les preuves générées par veriT sous plusieurs formes.

En particulier, nous avons ajouté :

- des règles de preuve pour les transformations qui n'ont pas produit de preuves, telles que la suppression de la double négation, et la suppression des littéraux répétés,
- de nouvelles règles pour différentes catégories de simplification effectuées lors du prétraitement,

- des coefficients avec des étapes arithmétiques linéaires, ce qui simplifie considérablement la vérification de ces étapes,
- le support de l'affichage de la substitution utilisée pendant l'instanciation du quantificateur,
- une règle de preuve pour exprimer la clausification des formules quantifiées,
- le support pour l'affichage des termes Skolem en tant que fonctions et une heuristique pour ne pas introduire de noms de termes superflus.

De plus, nous avons apporté de nombreuses améliorations à l'architecture de production des preuves, et corrigé de nombreux bogues.

Notre méthode de reconstruction augmente le taux de réussite de Sledgehammer en réduisant de moitié le taux d'échec et réduit le temps de vérification de 13%. Notre évaluation révèle que le temps d'exécution est influencé à la fois par les règles simples qui apparaissent très souvent, et par les règles complexes communes.

Améliorer la simplification des quantificateurs. La deuxième contribution est une nouvelle technique de simplification des quantificateurs. Elle est basée sur l'unification, et est le résultat de nombreuses tentatives visant à améliorer les capacités de raisonnement des quantificateurs de veriT, et de SMT en général. L'instanciation des quantificateurs est très sensible et les méthodes qui semblent bénéfiques d'un point de vue théorique s'avèrent néfastes en pratique. Néanmoins, la méthode développée fonctionne bien. Elle est le résultat de l'étude d'un exemple, de la mise en œuvre de la procédure la strictement nécessaire à la résolution de cet exemple, et d'une généralisation ultérieure.

On utilise des formules quantifiées valides pour remplacer certaines sous formules des formules complexes par des constants Booléennes. En ce faisant, nous obtenons des formules simplifiées qui sont ajoutés au problème. Nous proposons plusieurs variantes de la procédure de base.

L'idée de notre technique consiste à simplifier les formules quantifiées en remplaçant une sous-formule quantifiée par la constante Booléenne $\top$ ou $\bot$. Cela peut être fait si la matrice de cette sous-formule quantifiée peut être unifiée avec la matrice d'une formule unitaire, et est formalisée par la règle SUB.

Définition 1 (SUB Rule)

$$\frac{\forall x_1, \ldots, x_n.\, \psi_1 \qquad \forall x_{n+1}, \ldots, x_m.\, \varphi[Q\bar{y}.\, \psi_2]}{\forall x_{k_1}, \ldots, x_{k_j}.\, \varphi[b]\theta}\ \text{SUB}$$

où $Q \in \{\exists, \forall\}$. La sous-formule $Q\bar{y}.\, \psi_2$ n'apparaît que sous le quantificateur universel le plus externe de φ, et θ est une substitution. La règle est soumise aux conditions suivantes :

1. $\text{trim}(\psi_1)\theta = \text{trim}(\psi_2)\theta$, si $Q\bar{y}.\, \psi_2$ est faible;
2. $\text{trim}(\psi_1)\theta = \text{trim}(\text{sko}(Q\bar{y}.\, \psi_2, x_{n+1} \ldots x_m))\theta$, si $Q\bar{y}.\, \psi_2$ est forte;
3. les variables liées de la conclusion $\{x_{k_1}, x_{k_2}, \ldots, x_{k_j}\}$ sont exactement $\text{free}(\varphi[b]\theta)$;
4. $b = \top$ si $\text{pol}(\psi_1) = \text{pol}(\psi_2)$ et $b = \bot$ si $\text{pol}(\psi_1) \neq \text{pol}(\psi_2)$.

Le solveur SMT ajoute des formules dérivées de la règle SUB au problème d'entrée. Cette règle nous permet de combiner et de restreindre simplement la skolémisation, l'unification, et le remplacement de sous-formules avec la constante appropriée. La règle combine de façon correcte ces étapes sûres. D'abord, elle skolémise les variables $\bar{y}$. Ensuite, elle applique l'unificateur θ. La sous-formule de ψ_1 correspondant à $Q\bar{y}.\, \psi_2$ dans la règle de règle SUB est desormais équivalente à $\text{trim}(\psi_1)\theta$, et est remplacée par une constante Booléenne. La constante appropriée est choisie en fonction de la polarité des formules. Ce remplacement est correct puisque $\psi_1\theta$ est toujours vrai. Dans l'ensemble, la règle de SUB, ainsi que la procédure de simplification, ressemble quelque peu à la résolution unitaire où $\forall x_1, \ldots, x_n.\, \psi_1$ est la clause unitaire. Dans le cas des solveurs SMT, cependant, φ peut ne pas être une clause. De plus, ψ_1 et ψ_2 auront la structure complexe qui est préservée de l'entrée, puisque la plupart des techniques d'instanciation actuellement utilisées n'ont aucun avantage à appliquer la clausification complète et ne l'appliquent donc pas.

L'évaluation sur les benchmarks SMT-LIB montre que notre technique permet à veriT de résoudre des benchmarks jusqu'alors non résolus avec les autres stratégies. Lorsqu'elle est applicable, la méthode permet souvent à veriT de résoudre des problèmes dans un délai court. Les différentes variantes du processus de simplification sont utiles dans le cadre d'un schedule de stratégies.

À l'avenir, la méthode pourra également être utilisée comme une méthode de traitement en cours, affin de simplifier les formules existantes lorsque de nouvelles instances sont engendrées. Nous espérons également qu'elle pourra servir de premier tremplin pour une utilisation plus large des méthodes basées sur la résolution par un solveur SMT.

Planification de stratégie avec schedgen. La plupart des composants d'un solveur SMT peuvent être paramétrés et réglés avec précision. La sélection des bonnes valeurs pour les paramètres peut être difficile. Souvent, il n'y a pas de meilleur choix, et même s'il y en a un, des choix globaux non-optimaux peuvent être plus efficaces pour certains types de problèmes. Un choix spécifique de valeurs pour tous les paramètres exposés est une *stratégie*. Il est souvent crucial d'utiliser la bonne stratégie pour résoudre un problème dans un délai donné.

Puisque, pour de nombreux problèmes, il existe une stratégie permettant leur résolution en peu de temps, il est naturel d'essayer plusieurs stratégies. La façon la plus simple de le faire est d'essayer une liste de stratégies, une à une. Une méthode un peu plus sophistiquée consiste à préparer une liste de stratégies associée à un délai d'exécution : certaines stratégies peuvent être connues pour avoir des rendements décroissant si elles sont exécutées pendant une période plus longue. Nous appellerons une telle liste un *schedule*.

Nous avons développé une méthode simple basée sur l'optimisation linéaire en nombres entiers pour trouver un schedule qui est optimal par rapport au nombre de benchmarks résolus. Le développeur du solveur SMT doit définir une liste de stratégies et de délais autorisés, et doit enregistrer le temps que chaque stratégie prend pour résoudre les benchmarks de l'ensemble d'apprentissage. La méthode encode le problème de la recherche du schedule résolvant le plus grand nombre de benchmarks dans un délai global, comme un problème de programmation en nombres entiers. Globalement, la procédure de génération est conçue pour être facile à comprendre et éviter les surprises.

Cet encodage est implémenté dans une boîte à outils appelée schedgen pour générer des schedules. Elle est implémentée en Python et utilise la librairie PuLP [74] pour exprimer et résoudre les problèmes d'optimisation linéaire. La boîte à outils permet aux utilisateurs de générer des schedules simples et statiques et de les intégrer avec le solveur de leur choix. Outre

l'outil de base de génération de schedule, cette boîte à outils comprend également des outils auxiliaires qui permettent de simuler et de visualiser les schedules.

Nous avons évalué l'utilité pratique de schedgen. Pour ce faire, nous avons effectué une validation croisée avec cinq répétitions sur 9000 benchmarks de la suite SMT-LIB. Par conséquent, schedgen-finalize.py a été appelé cinq fois sur 7200 benchmarks pour générer des schedules. Chaque schedule a été évalué sur les 1800 benchmarks restants. En moyenne, les schedules générés ont résolu 1316 benchmarks, tandis qu'un schedule généré par une simple procédure gloutonne n'a résolu que 1312 benchmarks. Bien que l'augmentation des benchmarks résolus semble faible, la fenêtre d'amélioration ne représente qu'un petit pourcentage du nombre total de benchmarks : en moyenne, la meilleure stratégie seule résout déjà 1280 benchmarks, et même le meilleur solveur virtuel ne résout que 14 benchmarks de plus qu'un schedule glouton.

Conclusion. Les trois contributions de la thèse montrent que les solveurs SMT peuvent être adaptés à des applications spécifiques. Soit en les étendant, soit en fournissant de nouveaux outils et de nouvelles interfaces. Les solveurs SMT ne sont pas des boîtes noires fixes qui traite.

des problèmes logiques et produisent des réponses. Ce sont des systèmes logiciels complexes qui peuvent être adaptés. Malheureusement, cela nécessite une connaissance approfondie du solveur. De plus, malgré la flexibilité de l'intégration d'une théorie T dans CDCL(T), les utilisateurs de solveurs SMT développent rarement de nouvelles théories et de nouveaux solveurs de théories de manière indépendante. Cet écart entre le solveur et ses utilisateurs a déjà été identifié et caractérisé comme étant le « Strategy Challenge in SMT Solving » [48]. La communauté SMT devrait relever ce défi afin de renforcer la résolution de problèmes SMT.

Une façon de réduire l'écart entre le solveur et un utilisateur spécifique – les assistants de preuve – serait de passer à une logique d'ordre supérieur du côté du solveur SMT. Le solveur peut alors utiliser un encodage optimisé pour ses techniques de résolution, ou même utiliser des techniques de résolution adaptées à la logique d'ordre supérieur. Faire évoluer les prouveurs de théorèmes basés sur la superposition, ainsi que les solveurs SMT, vers la logique d'ordre supérieur était l'objectif principal du projet Matryoshka [25]. L'approche adoptée par le projet Matryoshka pour y parvenir consiste en une architecture stratifiée : le raisonnement du premier ordre reste intact, mais est étendu avec des règles pour raisonner avec des constructions d'ordre

supérieur. Du côté SMT, cela peut être réalisé en étendant l'instanciation des procédures [16].

Nous pouvons nous inspirer de cette approche pour construire des architectures qui offrent plus de contrôle aux utilisateurs. L'architecture sous-jacente au solveur basé sur CDCL(T) reste en grande partie inchangée, mais elle est soigneusement étendue avec des inférences contrôlées par l'utilisateur. Bien que les solveurs modernes ne supportent pas une telle approche, elle existe déjà officieusement grâce à l'instanciation basée sur des *triggers*. En choisissant soigneusement les triggers pour obtenir certains effets, les utilisateurs peuvent contrôler le processus de raisonnement [43]. Par exemple, si un problème encode des listes, le problème pourrait contenir le lemme $(\forall x\ xs.\ (head\ (cons\ x\ xs)) \approx x)$. Pour s'assurer que la fonction $head$ est appliquée lors de l'instanciation du quantificateur, l'utilisateur ajoutera le trigger $(head\ (cons\ x\ xs))$. En fait, l'utilisateur peut savoir qu'il n'est judicieux d'instancier ce lemme que lorsque le trigger correspond. Actuellement, cela ne peut pas être imposé sans désactiver toutes les autres méthodes d'instanciation. Une extension soigneusement conçue des triggers peut fournir plus de contrôle à l'utilisateur. Une telle extension spéculative pourrait être basée sur le domaine bien étudié de la réécriture de termes. Au lieu de triggers, l'utilisateur fournit des règles de réécriture ainsi qu'une description de comment et quand la règle doit être appliquée. Dans l'exemple ci-dessus, l'utilisateur ajouterait la règle de réécriture suivante $(head\ (cons\ x\ xs)) \mapsto x$. Même le développeur du solveur peut profiter d'une telle extension. Puisqu'il est facile de fournir des preuves pour l'application des règles de réécriture, l'extension peut être utilisée pour implémenter un prétraitement produisant des preuves ainsi que le raisonnement théorique.

Dans l'ensemble, la logique d'ordre supérieur fournira à l'utilisateur l'expressivité nécessaire pour encoder directement ses problèmes. Le mécanisme d'extension basé sur la réécriture et les théories intégrées fournira à l'utilisateur un mécanisme pour exprimer sa connaissance du domaine et pour assurer que le solveur peut résoudre ses problèmes.

The Alethe Proof Rules

This section is based on an informally published reference of the Alethe proof format. It was originally written by me for the work in Section 3.2. Later Haniel Barbosa, Mathias Fleury, Pascal Fontaine contributed.

This appendix provides a list of all proof rules supported by Alethe. To make this long list more accessible, this section first lists multiple classes of proof rules. The classes are not mutually exclusive: for example, the la_generic rule is both a linear arithmetic rule and introduces a tautology. The number in brackets is the position of the rule in the overall list of proof rules. Table 1 lists rules that serve a special purpose. Table 3 lists all rules that introduce tautologies. That is, regular rules that do not use premises.

The subsequent section, starting at page 167, defines all rules and provides example proofs for complicated rules. The index of proof rules at page 191 can be used to quickly find the definition of rules.

Classifications of the Rules

Rule	Description
assume (1)	Repetition of an input assumption.
hole (2)	Placeholder for rules not defined here.
subproof (10)	Concludes a subproof and discharges local assumptions.

Table 1 Special rules.

Rule	Description
resolution (7)	Chain resolution of two or more clauses.
th_resolution (6)	As resolution, but used by theory solvers.
tautology (8)	Simplification of tautological clauses to $\top$.
contraction (9)	Removal of duplicated literals.

Table 2 Resolution and related rules.

Rule	Conclusion
true (3)	$\top$
false (4)	$\neg\bot$
not_not (5)	$\neg(\neg\neg\varphi), \varphi$
la_generic (11)	Tautologous disjunction of linear inequalities.
lia_generic (12)	Tautologous disjunction of linear integer inequalities.
la_disequality (13)	$t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
la_totality (14)	$t_1 \leq t_2 \vee t_2 \leq t_1$
la_tautology (15)	A trivial linear tautology.
forall_inst (19)	Quantifier instantiation.
refl (20)	Reflexivity after applying the context.
eq_reflexive (23)	$t \approx t$ without context.
eq_transitive (24)	$\neg(t_1 \approx t_2), \ldots, \neg(t_{n-1} \approx t_n), t_1 \approx t_n$
eq_congruent (25)	$\neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)$
eq_congruent_pred (26)	$\neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), P(t_1, \ldots, t_n) \approx P(u_1, \ldots, u_n)$
qnt_cnf (27)	Clausification of a quantified formula.
and_pos (43)	$\neg(\varphi_1 \wedge \ldots \wedge \varphi_n), \varphi_k$
and_neg (44)	$(\varphi_1 \wedge \ldots \wedge \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$
or_pos (45)	$\neg(\varphi_1 \vee \ldots \vee \varphi_n), \varphi_1, \ldots, \varphi_n$
or_neg (46)	$(\varphi_1 \vee \ldots \vee \varphi_n), \neg\varphi_k$
xor_pos1 (47)	$\neg(\varphi_1 \mathbf{xor} \varphi_2), \varphi_1, \varphi_2$
xor_pos2 (48)	$\neg(\varphi_1 \mathbf{xor} \varphi_2), \neg\varphi_1, \neg\varphi_2$
xor_neg1 (49)	$\varphi_1 \mathbf{xor} \varphi_2, \varphi_1, \neg\varphi_2$
xor_neg2 (50)	$\varphi_1 \mathbf{xor} \varphi_2, \neg\varphi_1, \varphi_2$
implies_pos (51)	$\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$

Table 3a Rules introducing tautologies.

Rule	Conclusion
implies_neg1 (52)	$\varphi_1 \to \varphi_2, \varphi_1$
implies_neg2 (53)	$\varphi_1 \to \varphi_2, \neg\varphi_2$
equiv_pos1 (54)	$\neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2$
equiv_pos2 (55)	$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$
equiv_neg1 (56)	$\varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2$
equiv_neg2 (57)	$\varphi_1 \approx \varphi_2, \varphi_1, \varphi_2$
ite_pos1 (60)	$\neg(\mathbf{ite}\varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \varphi_3$
ite_pos2 (61)	$\neg(\mathbf{ite}\varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \varphi_2$
ite_neg1 (62)	$\mathbf{ite}\varphi_1\ \varphi_2\ \varphi_3, \varphi_1, \neg\varphi_3$
ite_neg2 (63)	$\mathbf{ite}\varphi_1\ \varphi_2\ \varphi_3, \neg\varphi_1, \neg\varphi_2$
connective_def (66)	Definition of the Boolean connectives.
and_simplify (67)	Simplification of a conjunction.
or_simplify (68)	Simplification of a disjunction.
not_simplify (69)	Simplification of a Boolean negation.
implies_simplify (70)	Simplification of an implication.
equiv_simplify (71)	Simplification of an equivalence.
bool_simplify (72)	Simplification of combined Boolean connectives.
ac_simp (73)	Flattening of nested $\vee$ or $\wedge$.
ite_simplify (74)	Simplification of if-then-else.
qnt_simplify (75)	Simplification of constant quantified formulas.
qnt_join (77)	Joining of consecutive quantifiers.
qnt_rm_unused (78)	Removal of unused quantified variables.
eq_simplify (79)	Simplification of equality.
div_simplify (80)	Simplification of division.
prod_simplify (81)	Simplification of products.
unary_minus_simplify (82)	Simplification of the unary minus.

Table 3b Rules introducing tautologies.

Rule	Conclusion
minus_simplify (83)	Simplification of subtractions.
sum_simplify (84)	Simplification of sums.
comp_simplify (85)	Simplification of arithmetic comparisons.
distinct_elim (87)	Elimination of the distinction predicate.
la_rw_eq (88)	$(t \approx u) \approx (t \leq u \wedge u \leq t)$
nary_elim (89)	Replace n-ary operators with binary application.

Table 3c Rules introducing tautologies.

Rule	Conclusion
la_generic (11)	Tautologous disjunction of linear inequalities.
lia_generic (12)	Tautologous disjunction of linear integer inequalities.
la_disequality (13)	$t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
la_totality (14)	$t_1 \leq t_2 \vee t_2 \leq t_1$
la_tautology (15)	A trivial linear tautology.
la_rw_eq (88)	$(t \approx u) \approx (t \leq u \wedge u \leq t)$
div_simplify (80)	Simplification of division.
prod_simplify (81)	Simplification of products.
unary_minus_simplify (82)	Simplification of the unary minus.
minus_simplify (83)	Simplification of subtractions.
sum_simplify (84)	Simplification of sums.
comp_simplify (85)	Simplification of arithmetic comparisons.

Table 4 Linear arithmetic rules.

Rule	Conclusion
forall_inst (19)	Instantiation of a universal variable.
bind (16)	Renaming of bound variables.
sko_ex (17)	Skolemization of existential variables.
sko_forall (18)	Skolemization of universal variables.
qnt_cnf (27)	Clausification of quantified formulas.
qnt_simplify (75)	Simplification of constant quantified formulas.
onepoint (76)	The one-point rule.
qnt_join (77)	Joining of consecutive quantifiers.
qnt_rm_unused (78)	Removal of unused quantified variables.

Table 5 Quantifier handling.

Rule	Conclusion
sko_ex (17)	Skolemization of existential variables.
sko_forall (18)	Skolemization of universal variables.

Table 6 Skolemization rules.

Rule	Conclusion
and (28)	And elimination.
not_or (29)	Elimination of a negated disjunction.
or (30)	Disjunction to clause.
not_and (31)	Distribution of negation over a conjunction.
xor1 (32)	From $\mathbf{xor}\ \varphi_1 \varphi_2$ to φ_1, φ_2.
xor2 (33)	From $\mathbf{xor}\ \varphi_1 \varphi_2$ to $\neg\varphi_1, \neg\varphi_2$.
not_xor1 (34)	From $\neg(\mathbf{xor}\ \varphi_1 \varphi_2)$ to $\varphi_1, \neg\varphi_2$.
not_xor2 (35)	From $\neg(\mathbf{xor}\ \varphi_1 \varphi_2)$ to $\neg\varphi_1, \varphi_2$.
implies (36)	From $\varphi_1 \to \varphi_2$ to $\neg\varphi_1, \varphi_2$.
not_implies1 (37)	From $\neg(\varphi_1 \to \varphi_2)$ to φ_1.
not_implies2 (38)	From $\neg(\varphi_1 \to \varphi_2)$ to $\neg\varphi_2$.

Table 7a Clausification rules. These rules can be used to perform propositional clausification.

Rule	Conclusion
equiv1 (39)	From $\varphi_1 \approx \varphi_2$ to $\neg\varphi_1, \varphi_2$.
equiv2 (40)	From $\varphi_1 \approx \varphi_2$ to $\varphi_1, \neg\varphi_2$.
not_equiv1 (41)	From $\neg(\varphi_1 \approx \varphi_2)$ to φ_1, φ_2.
not_equiv2 (42)	From $\neg(\varphi_1 \approx \varphi_2)$ to $\neg\varphi_1, \neg\varphi_2$.
and_pos (43)	$\neg(\varphi_1 \wedge \ldots \wedge \varphi_n), \varphi_k$
and_neg (44)	$(\varphi_1 \wedge \ldots \wedge \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$
or_pos (45)	$\neg(\varphi_1 \vee \ldots \vee \varphi_n), \varphi_1, \ldots, \varphi_n$
or_neg (46)	$(\varphi_1 \vee \ldots \vee \varphi_n), \neg\varphi_k$
xor_pos1 (47)	$\neg(\varphi_1 \textbf{ xor } \varphi_2), \varphi_1, \varphi_2$
xor_pos2 (48)	$\neg(\varphi_1 \textbf{ xor } \varphi_2), \neg\varphi_1, \neg\varphi_2$
xor_neg1 (49)	$\varphi_1 \textbf{ xor } \varphi_2, \varphi_1, \neg\varphi_2$
xor_neg2 (50)	$\varphi_1 \textbf{ xor } \varphi_2, \neg\varphi_1, \varphi_2$
implies_pos (51)	$\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$
implies_neg1 (52)	$\varphi_1 \rightarrow \varphi_2, \varphi_1$
implies_neg2 (53)	$\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$
equiv_pos1 (54)	$\neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2$
equiv_pos2 (55)	$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$
equiv_neg1 (56)	$\varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2$
equiv_neg2 (57)	$\varphi_1 \approx \varphi_2, \varphi_1, \varphi_2$
let (86)	Elimination of the **let** operator.
distinct_elim (87)	Elimination of the **distinct** operator.
nary_elim (89)	Elimination of n-ary application of operators.

Table 7b Clausification rules. These rules can be used to perform propositional clausification.

Rule	Conclusion
connective_def (66)	Definition of the Boolean connectives.
and_simplify (67)	Simplification of a conjunction.
or_simplify (68)	Simplification of a disjunction.
not_simplify (69)	Simplification of a Boolean negation.
implies_simplify (70)	Simplification of an implication.
equiv_simplify (71)	Simplification of an equivalence.
bool_simplify (72)	Simplification of combined Boolean connectives.
ac_simp (73)	Simplification of nested disjunctions and conjunctions.
ite_simplify (74)	Simplification of if-then-else.
qnt_simplify (75)	Simplification of constant quantified formulas.
onepoint (76)	The one-point rule.
qnt_join (77)	Joining of consecutive quantifiers.
qnt_rm_unused (78)	Removal of unused quantified variables.
eq_simplify (79)	Simplification of equalities.
div_simplify (80)	Simplification of division.
prod_simplify (81)	Simplification of products.
unary_minus_simplify (82)	Simplification of the unary minus.
minus_simplify (83)	Simplification of subtractions.
sum_simplify (84)	Simplification of sums.
comp_simplify (85)	Simplification of arithmetic comparisons.
qnt_simplify (75)	Simplification of constant quantified formulas.

Table 8 Simplification rules. These rules represent typical operator-level simplifications.

Rule List

Rule 1: assume

$i.\ \triangleright \qquad \varphi \qquad$ assume

where φ corresponds up to the orientation of equalities to a formula asserted in the input problem.

Remark. This rule can not be used by the (`step` ...) command. Instead it corresponds to the dedicated (`assume` ...) command.

Rule 2: hole

$i.\ \triangleright \qquad \varphi \qquad$ (hole p_1, ..., p_n) [a_1, ..., a_n]

where φ is any well-formed formula.

This rule can be used to express holes in the proof. It can be used by solvers as a placeholder for proof steps that are not yet expressed by the proof rules in this document. A proof checker *must not* accept a proof as valid that contains this rule even if the checker can somehow check this rule. However, it is possible for checkers to have a dedicated status for proofs that contain this rule and are otherwise valid. Any other tool can accept or reject proofs that contain this rule.

The premises and arguments are arbitrary, but must follow the syntax for premises and arguments.

Rule 3: true

$i.\ \triangleright \qquad \top \qquad$ true

Rule 4: false

$i.\ \triangleright \qquad \bot \qquad$ false

Rule 5: not_not

$i.\ \triangleright \qquad \neg(\neg\neg\varphi), \varphi \qquad$ not_not

Remark. This rule is useful to remove double negations from a clause by resolving a clause with the double negation on φ.

Rule 6: th_resolution This rule is the resolution of two or more clauses.

$$\begin{array}{lll} i_1. \triangleright & l_1^1, \ldots, l_{k^1}^1 & (\ldots) \\ & \vdots & \\ i_n. \triangleright & l_1^n, \ldots, l_{k^n}^n & (\ldots) \\ j. \triangleright & l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m} & (\text{th\_resolution } i_1, \ldots, i_n) \end{array}$$

where $l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m}$ are from l_j^i and are the result of a chain of predicate resolution steps on the clauses of i_1 to i_n. It is possible that $m = 0$, i.e. that the result is the empty clause. When performing resolution steps, the rule implicitly merges repeated negations at the start of the formulas l_j^i. For example, the formulas $\neg\neg\neg P$ and $\neg\neg P$ can serve as pivots during resolution. The first formula is interpreted as $\neg P$ and the second as just P for the purpose of performing resolution steps.

This rule is only used when the resolution step is not emitted by the SAT solver. See the equivalent resolution rule for the rule emitted by the SAT solver.

Remark. The definition given here is very general. The motivation for this to ensure the definition covers all possible resolution steps generated by the existing proof generation code in veriT. It will be restricted after a full review of the code. As a consequence of this checking this rule is theoretically NP-complete. In practice, however, the th_resolution-steps produced by veriT are simple. Experience with reconstructing the step in Isabelle/HOL shows that checking can done by naive decision procedures. The vast majority of th_resolution-steps are binary resolution steps.

Rule 7: resolution This rule is equivalent to the th_resolution rule, but it is emitted by the SAT solver instead of theory reasoners. The differentiation serves only informational purpose.

Rule 8: tautology

$$\begin{array}{lll} i. \triangleright & l_1, \ldots, l_k, \ldots, l_m, \ldots, l_n & (\ldots) \\ j. \triangleright & \top & (\text{tautology } i) \end{array}$$

and l_k, l_m are such that

$$l_k = \underbrace{\neg \ldots \neg}_{o} \varphi$$
$$l_m = \underbrace{\neg \ldots \neg}_{p} \varphi$$

and one of o, p is odd and the other even. Either can be 0.

Rule 9: contraction

$$\begin{array}{lll} i. \rhd & l_1, \ldots, l_n & (\ldots) \\ j. \rhd & l_{k_1}, \ldots, l_{k_m} & (\text{contraction } i) \end{array}$$

where $m \leq n$ and $k_1 \ldots k_m$ is a monotonic map to $1 \ldots n$ such that $l_{k_1} \ldots l_{k_m}$ are pairwise distinct and $\{l_1, \ldots, l_n\} = \{l_{k_1}, \ldots, l_{k_m}\}$. Hence, this rule removes duplicated literals.

Rule 10: subproof The subproof rule completes a subproof and discharges local assumptions. Every subproof starts with some assume steps. The last step of the subproof is the conclusion.

$$\begin{array}{lll|l} i_1. \rhd & \varphi_1 & & \text{assume} \\ & \vdots & & \\ i_n. \rhd & \varphi_n & & \text{assume} \\ & \vdots & & \\ j. \rhd & \psi & & (\ldots) \\ \hline \text{k.} \rhd & \neg\varphi_1, \ldots, \neg\varphi_n, \psi & & \text{subproof} \end{array}$$

Rule 11: la_generic A step of the la_generic rule represents a tautological clause of linear disequalities. It can be checked by showing that the conjunction of the negated disequalities is unsatisfiable. After the application of some strengthening rules, the resulting conjunction is unsatisfiable, even if integer variables are assumed to be real variables.

A linear inequality is of term of the form

$$\sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2$$

where $\bowtie \in \{\approx, <, >, \leq, \geq\}$, where $m \geq n$, c_i, d_1, d_2 are either integer or real constants, and for each i c_i and t_i have the same sort. We will write $s_1 \bowtie s_2$.

Let $l_1, \ldots, l_n$ be linear inequalities and $a_1, \ldots, a_n$ rational numbers, then a la_generic step has the form

$$i.\ \triangleright \qquad \varphi_1, \ldots, \varphi_o \qquad \text{la\_generic } [a_1, \ldots, a_o]$$

where φ_i is either $\neg l_i$ or l_i, but never $s_1 \approx s_2$.

If the current theory does not have rational numbers, then the a_i are printed using integer division. They should, nevertheless, be interpreted as rational numbers. If d_1 or d_2 are 0, they might not be printed.

To check the unsatisfiability of the negation of $\varphi_1 \vee \ldots \vee \varphi_o$ one performs the following steps for each literal. For each i, let $\varphi := \varphi_i$ and $a := a_i$.

1. If $\varphi = s_1 > s_2$, then let $\varphi := s_1 \leq s_2$. If $\varphi = s_1 \geq s_2$, then let $\varphi := s_1 < s_2$. If $\varphi = s_1 < s_2$, then let $\varphi := s_1 \geq s_2$. If $\varphi = s_1 \leq s_2$, then let $\varphi := s_1 > s_2$. This negates the literal.
2. If $\varphi = \neg(s_1 \bowtie s_2)$, then let $\varphi := s_1 \bowtie s_2$.
3. Replace φ by $\sum_{i=0}^{n} c_i \times t_i - \sum_{i=n+1}^{m} c_i \times t_i \bowtie d$ where $d := d_2 - d_1$.
4. Now φ has the form $s_1 \bowtie d$. If all variables in s_1 are integer sorted: replace $\bowtie d$ according to Table 9.
5. If $\bowtie$ is $\approx$ replace l by $\sum_{i=0}^{m} a \times c_i \times t_i \approx a \times d$, otherwise replace it by $\sum_{i=0}^{m} |a| \times c_i \times t_i \approx |a| \times d$.

$\bowtie$	If d is an integer	Otherwise
$>$	$\geq d + 1$	$\geq \lfloor d \rfloor + 1$
$\geq$	$\geq d$	$\geq \lfloor d \rfloor + 1$

Table 9 Strengthening rules for la_generic.

Finally, the sum of the resulting literals is trivially contradictory. The sum

$$\sum_{k=1}^{o} \sum_{i=1}^{m^o} c_i^k * t_i^k \bowtie \sum_{k=1}^{o} d^k$$

where c_i^k is the constant c_i of literal l_k, t_i^k is the term t_i of l_k, and d^k is the constant d of l_k. The operator $\bowtie$ is $\approx$ if all operators are $\approx$, $>$ if all are either

$\approx$ or $>$, and $\geq$ otherwise. The a_i must be such that the sum on the left-hand side is 0 and the right-hand side is > 0 (or ≥ 0 if $\bowtie$ is $>$).

Example 11.1
A simple la_generic step in the logic LRA might look like this:

```
(step t10 (cl (not (> (f a) (f b))) (not (= (f a) (f b))))
    :rule la_generic :args (1.0 (- 1.0)))
```

To verify this we have to check the insatisfiability of $(f\ a) > (f\ b) \wedge (f\ a) \approx (f\ b)$ (step 2). After step 3 we get $(f\ a) - (f\ b) > 0 \wedge (f\ a) - (f\ b) \approx 0$. Since we don't have an integer sort in this logic step 4 does not apply. Finally, after step 5 the conjunction is $(f\ a) - (f\ b) > 0 \wedge -(f\ a) + (f\ b) \approx 0$. This sums to $0 > 0$, which is a contradiction.

Example 11.2
The following la_generic step is from a QF_UFLIA problem:

```
(step t11 (cl (not (<= f3 0)) (<= (+ 1 (* 4 f3)) 1))
    :rule la_generic :args (1 (div 1 4)))
```

After normalization we get $-f_3 \geq 0 \wedge 4 \times f_3 > 0$. This time step 4 applies and we can strengthen this to $-f_3 \geq 0 \wedge 4 \times f_3 \geq 1$ and after multiplication we get $-f_3 \geq 0 \wedge f_3 \geq ¼$. Which sums to the contradiction $¼ \geq 0$.

Rule 12: lia_generic This rule is a placeholder rule for integer arithmetic solving. It takes the same form as la_generic, without the additional arguments.

$$i.\ \triangleright \qquad \varphi_1, \ldots, \varphi_o \qquad \text{(lia\_generic)}$$

with φ_i being linear inequalities. The disjunction $\varphi_1 \vee \ldots \vee \varphi_n$ is a tautology in the theory of linear integer arithmetic.

Remark. Since this rule can introduce a disjunction of arbitrary linear integer inequalities without any additional hints, proof checking can be NP-hard. Hence, this rule should be avoided when possible.

Rule 13: la_disequality

$$i.\ \triangleright \qquad t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1) \qquad \text{(la\_disequality)}$$

Rule 14: la_totality

$$i.\ \triangleright \qquad t_1 \leq t_2 \vee t_2 \leq t_1 \qquad \text{(la\_totality)}$$

Rule 15: la_tautology This rule is a linear arithmetic tautology which can be checked without sophisticated reasoning. It has either the form

$$i.\ \triangleright \qquad \varphi \qquad \text{(la\_tautology)}$$

where φ is either a linear inequality $s_1 \bowtie s_2$ or $\neg(s_1 \bowtie s_2)$. After performing step 1 to 3 of the process for checking the la_generic the result is trivially unsatisfiable.

The second form handles bounds on linear combinations. It is binary clause:

$$i.\ \triangleright \qquad \varphi_1 \vee \varphi_2 \qquad \text{(la\_tautology)}$$

It can be checked by using the procedure for la_generic with while setting the arguments to 1. Informally, the rule follows one of several cases:

- $\neg(s_1 \leq d_1) \vee s_1 \leq d_2$ where $d_1 \leq d_2$
- $s_1 \leq d_1 \vee \neg(s_1 \leq d_2)$ where $d_1 = d_2$
- $\neg(s_1 \geq d_1) \vee s_1 \geq d_2$ where $d_1 \geq d_2$
- $s_1 \geq d_1 \vee \neg(s_1 \geq d_2)$ where $d_1 = d_2$
- $\neg(s_1 \leq d_1) \vee \neg(s_1 \geq d_2)$ where $d_1 < d_2$

The inequalities $s_1 \bowtie d$ are are the result of applying normalization as for the rule la_generic.

Rule 16: bind The bind rule is used to rename bound variables.

$$\begin{array}{lll|l} & & & \vdots & \\ j. & \Gamma, y_1, \ldots, y_n, x_1 \mapsto y_1, \ldots, x_n \mapsto y_n \triangleright & & \varphi \approx \varphi' & (\ldots) \\ \hline k. & & \triangleright & \forall x_1, \ldots, x_n.\varphi \approx \forall y_1, \ldots, y_n.\varphi' & \text{bind} \end{array}$$

where the variables $y_1, \ldots, y_n$ is not free in $\forall x_1, \ldots, x_n.\varphi$.

Rule 17: sko_ex The sko_ex rule skolemizes existential quantifiers.

$$\begin{array}{lll|l|l} & & & \vdots & \\ j. & \Gamma, x_1 \mapsto \varepsilon_1, \ldots, x_n \mapsto \varepsilon_n \rhd & & \varphi \approx \psi & (\ldots) \\ \hline k. & & \rhd & \exists x_1, \ldots, x_n.\varphi \approx \psi & \text{sko\_ex} \end{array}$$

where ε_i stands for $\varepsilon x_i.(\exists x_{i+1}, \ldots, x_n.\varphi[x_1 \mapsto \varepsilon_1, \ldots, x_{i-1} \mapsto \varepsilon_{i-1}])$.

Rule 18: sko_forall The sko_forall rule skolemizes universal quantifiers.

$$\begin{array}{lll|l|l} & & & \vdots & \\ j. & \Gamma, x_1 \mapsto (\varepsilon x_1.\neg\varphi), \ldots, x_n \mapsto (\varepsilon x_n.\neg\varphi) \rhd & & \varphi \approx \psi & (\ldots) \\ \hline k. & & \rhd & \forall x_1, \ldots, x_n.\varphi \approx \psi & \text{sko\_forall} \end{array}$$

Rule 19: forall_inst

$$i. \ \rhd \ \neg(\forall x_1, \ldots, x_n.P) \vee P[x_1 \mapsto t_1]\ldots[x_n \mapsto t_n] \quad \text{forall\_inst } [(x_{k_1}, t_{k_1}), \ldots, (x_{k_n}, t_{k_n})]$$

where $k_1, \ldots, k_n$ is a permutation of $1, \ldots, n$ and x_i and k_i have the same sort. The arguments (x_{k_i}, t_{k_i}) are printed as `(:= xki tki)`.

Remark. In Section 4.6 we discuss an alternative quantifier instantiation rule. The resulting proof would be more fine grained and this would also be an opportunity to provide a proof for the clausification as currently done by qnt_cnf.

Rule 20: refl

$$j. \ \rhd \Gamma \qquad t_1 \approx t_2 \qquad \text{refl}$$

where, if $\sigma = \text{subst}(\Gamma)$, the terms $t_1\sigma$ and t_2 are syntactically equal up to the orientation of equalities.

Remark. This is the only rule that requires the application of the context.

Rule 21: trans

$$\begin{array}{lcl} i_1.\ \triangleright\ \Gamma & t_1 \approx t_2 & (\ldots) \\ i_2.\ \triangleright\ \Gamma & t_2 \approx t_3 & (\ldots) \\ & \vdots & \\ i_n.\ \triangleright\ \Gamma & t_n \approx t_{n+1} & (\ldots) \\ j.\ \triangleright\ \Gamma & t_1 \approx t_{n+1} & (\text{trans}\ \ i_1, \ldots, i_n) \end{array}$$

Rule 22: cong

$$\begin{array}{lcl} i_1.\ \triangleright\ \Gamma & t_1 \approx u_1 & (\ldots) \\ i_2.\ \triangleright\ \Gamma & t_2 \approx u_2 & (\ldots) \\ & \vdots & \\ i_n.\ \triangleright\ \Gamma & t_n \approx u_n & (\ldots) \\ j.\ \triangleright\ \Gamma & (f\ t_1 \cdots t_n) \approx (f\ u_1 \cdots u_n) & (\text{cong}\ \ i_1, \ldots, i_n) \end{array}$$

where f is any n-ary function symbol of appropriate sort.

Rule 23: eq_reflexive

$$i.\ \triangleright \qquad t \approx t \qquad \text{eq\_reflexive}$$

Rule 24: eq_transitive

$$i.\ \triangleright \qquad \neg(t_1 \approx t_2), \ldots, \neg(t_{n-1} \approx t_n), t_1 \approx t_n \qquad \text{eq\_transitive}$$

Rule 25: eq_congruent

$$i.\ \triangleright \qquad \neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), (f\ t_1 \cdots t_n) \approx (f\ u_1 \cdots u_n) \qquad \text{eq\_congruent}$$

Rule 26: eq_congruent_pred

$i.\ \triangleright\ \neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), (P\ t_1 \cdots t_n) \approx (P\ u_1 \cdots u_n)$ eq_congruent_pred

where P is a function symbol with co-domain sort **Bool**.

Rule 27: qnt_cnf

$i.\ \triangleright \qquad \neg(\forall x_1, \ldots, x_n.\varphi) \vee \forall x_{k_1}, \ldots, x_{k_m}.\varphi' \qquad$ qnt_cnf

This rule expresses clausification of a term under a universal quantifier. This is used by conflicting instantiation. φ' is one of the clause of the clause normal form of φ. The variables $x_{k_1}, \ldots, x_{k_m}$ are a permutation of $x_1, \ldots, x_n$ plus additional variables added by prenexing φ. Normalization is performed in two phases. First, the negative normal form is formed, then the result is prenexed. The result of the first step is $\Phi(\varphi, 1)$ where:

$$
\begin{aligned}
\Phi(\neg\varphi, 1) &:= \Phi(\varphi, 0)\\
\Phi(\neg\varphi, 0) &:= \Phi(\varphi, 1)\\
\Phi(\varphi_1 \vee \ldots \vee \varphi_n, 1) &:= \Phi(\varphi_1, 1) \vee \ldots \vee \Phi(\varphi_n, 1)\\
\Phi(\varphi_1 \wedge \ldots \wedge \varphi_n, 1) &:= \Phi(\varphi_1, 1) \wedge \ldots \wedge \Phi(\varphi_n, 1)\\
\Phi(\varphi_1 \vee \ldots \vee \varphi_n, 0) &:= \Phi(\varphi_1, 0) \wedge \ldots \wedge \Phi(\varphi_n, 0)\\
\Phi(\varphi_1 \wedge \ldots \wedge \varphi_n, 0) &:= \Phi(\varphi_1, 0) \vee \ldots \vee \Phi(\varphi_n, 0)\\
\Phi(\varphi_1 \to \varphi_2, 1) &:= (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_2, 0) \vee \Phi(\varphi_1, 1))\\
\Phi(\varphi_1 \to \varphi_2, 0) &:= (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_2, 1) \wedge \Phi(\varphi_1, 0))\\
\Phi(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3, 1) &:= (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_1, 1) \vee \Phi(\varphi_3, 1))\\
\Phi(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3, 0) &:= (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_1, 0) \wedge \Phi(\varphi_3, 0))\\
\Phi(\forall x_1, \ldots, x_n.\varphi, 1) &:= \forall x_1, \ldots, x_n.\Phi(\varphi, 1)\\
\Phi(\exists x_1, \ldots, x_n.\varphi, 1) &:= \exists x_1, \ldots, x_n.\Phi(\varphi, 1)\\
\Phi(\forall x_1, \ldots, x_n.\varphi, 0) &:= \exists x_1, \ldots, x_n.\Phi(\varphi, 0)\\
\Phi(\exists x_1, \ldots, x_n.\varphi, 0) &:= \forall x_1, \ldots, x_n.\Phi(\varphi, 0)\\
\Phi(\varphi, 1) &:= \varphi\\
\Phi(\varphi, 0) &:= \neg\varphi
\end{aligned}
$$

Remark. This is a placeholder rule that combines the many steps done during clausification.

Rule 28: and

$i.\ \triangleright$	$\varphi_1 \land \cdots \land \varphi_n$	$(\ldots)$
$j.\ \triangleright$	φ_k	(and i)

and $1 \leq k \leq n$.

Rule 29: not_or

$i.\ \triangleright$	$\neg(\varphi_1 \lor \cdots \lor \varphi_n)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_k$	(not_or i)

and $1 \leq k \leq n$.

Rule 30: or

$i.\ \triangleright$	$\varphi_1 \lor \cdots \lor \varphi_n$	$(\ldots)$
$j.\ \triangleright$	$\varphi_1, \ldots, \varphi_n$	(or i)

Remark. This rule deconstructs the `or` operator into a clause denoted by `cl`.

Example 30.1
An application of the or rule.

```
(step t15 (cl (or (= a b) (not (<= a b)) (not (<= b a))))
    :rule la_disequality)
(step t16 (cl     (= a b) (not (<= a b)) (not (<= b a)))
    :rule or :premises (t15))
```

Rule 31: not_and

$i.\ \triangleright$	$\neg(\varphi_1 \land \ldots \land \varphi_n)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \ldots, \neg\varphi_n$	(not_and i)

Rule 32: xor1

$i.\ \triangleright$	$\mathsf{xor}\ \varphi_1\ \varphi_2$	$(\ldots)$
$j.\ \triangleright$	φ_1, φ_2	(xor1 i)

Rule 33: xor2

$i.\ \triangleright$	$\mathbf{xor}\ \varphi_1\ \varphi_2$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \neg\varphi_2$	(xor2 i)

Rule 34: not_xor1

$i.\ \triangleright$	$\neg(\mathbf{xor}\ \varphi_1\ \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	$\varphi_1, \neg\varphi_2$	(not_xor1 i)

Rule 35: not_xor2

$i.\ \triangleright$	$\neg(\mathbf{xor}\ \varphi_1\ \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \varphi_2$	(not_xor2 i)

Rule 36: implies

$i.\ \triangleright$	$\varphi_1 \to \varphi_2$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \varphi_2$	(implies i)

Rule 37: not_implies1

$i.\ \triangleright$	$\neg(\varphi_1 \to \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	φ_1	(not_implies1 i)

Rule 38: not_implies2

$i.\ \triangleright$	$\neg(\varphi_1 \to \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_2$	(not_implies2 i)

Rule 39: equiv1

$i.\ \triangleright$	$\varphi_1 \approx \varphi_2$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \varphi_2$	(equiv1 i)

Rule 40: equiv2

$i.\ \triangleright$	$\varphi_1 \approx \varphi_2$	$(\ldots)$
$j.\ \triangleright$	$\varphi_1, \neg\varphi_2$	(equiv2 i)

Rule 41: not_equiv1

$i.\ \triangleright$	$\neg(\varphi_1 \approx \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	φ_1, φ_2	(not_equiv1 i)

Rule 42: not_equiv2

$i.\ \triangleright$	$\neg(\varphi_1 \approx \varphi_2)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \neg\varphi_2$	(not_equiv2 i)

Rule 43: and_pos

$i.\ \triangleright$	$\neg(\varphi_1 \wedge \cdots \wedge \varphi_n), \varphi_k$	and_pos

with $1 \leq k \leq n$.

Rule 44: and_neg

$i.\ \triangleright$	$(\varphi_1 \wedge \cdots \wedge \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$	and_neg

Rule 45: or_pos

$i.\ \triangleright$	$\neg(\varphi_1 \vee \cdots \vee \varphi_n), \varphi_1, \ldots, \varphi_n$	or_pos

Rule 46: or_neg

$i.\ \triangleright$	$(\varphi_1 \vee \cdots \vee \varphi_n), \neg\varphi_k$	or_neg

with $1 \leq k \leq n$.

Rule 47: xor_pos1

$i.\ \triangleright$	$\neg(\varphi_1 \ \mathbf{xor}\ \varphi_2), \varphi_1, \varphi_2$	xor_pos1

Rule 48: xor_pos2

$i.\ \triangleright \qquad \neg(\varphi_1 \mathbin{\mathbf{xor}} \varphi_2), \neg\varphi_1, \neg\varphi_2 \qquad$ xor_pos2

Rule 49: xor_neg1

$i.\ \triangleright \qquad \varphi_1 \mathbin{\mathbf{xor}} \varphi_2, \varphi_1, \neg\varphi_2 \qquad$ xor_neg1

Rule 50: xor_neg2

$i.\ \triangleright \qquad \varphi_1 \mathbin{\mathbf{xor}} \varphi_2, \neg\varphi_1, \varphi_2 \qquad$ xor_neg2

Rule 51: implies_pos

$i.\ \triangleright \qquad \neg(\varphi_1 \to \varphi_2), \neg\varphi_1, \varphi_2 \qquad$ implies_pos

Rule 52: implies_neg1

$i.\ \triangleright \qquad \varphi_1 \to \varphi_2, \varphi_1 \qquad$ implies_neg1

Rule 53: implies_neg2

$i.\ \triangleright \qquad \varphi_1 \to \varphi_2, \neg\varphi_2 \qquad$ implies_neg2

Rule 54: equiv_pos1

$i.\ \triangleright \qquad \neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2 \qquad$ equiv_pos1

Rule 55: equiv_pos2

$i.\ \triangleright \qquad \neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2 \qquad$ equiv_pos2

Rule 56: equiv_neg1

$i.\ \triangleright \qquad \varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2 \qquad$ equiv_neg1

Rule 57: equiv_neg2

$i.\ \triangleright \qquad \varphi_1 \approx \varphi_2, \varphi_1, \varphi_2 \qquad$ equiv_neg2

Rule 58: ite1

$i.\ \triangleright$	$\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3$	$(\ldots)$
$j.\ \triangleright$	φ_1, φ_3	(ite1 i)

Rule 59: ite2

$i.\ \triangleright$	$\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \varphi_2$	(ite2 i)

Rule 60: ite_pos1

$i.\ \triangleright$	$\neg(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \varphi_3$	(ite_pos1)

Rule 61: ite_pos2

$i.\ \triangleright$	$\neg(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \varphi_2$	(ite_pos2)

Rule 62: ite_neg1

$i.\ \triangleright$	$\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3, \varphi_1, \neg\varphi_3$	(ite_neg1)

Rule 63: ite_neg2

$i.\ \triangleright$	$\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3, \neg\varphi_1, \neg\varphi_2$	(ite_neg2)

Rule 64: not_ite1

$i.\ \triangleright$	$\neg(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3)$	$(\ldots)$
$j.\ \triangleright$	$\varphi_1, \neg\varphi_3$	(not_ite1 i)

Rule 65: not_ite2

$i.\ \triangleright$	$\neg(\mathsf{ite}\ \varphi_1\ \varphi_2\ \varphi_3)$	$(\ldots)$
$j.\ \triangleright$	$\neg\varphi_1, \neg\varphi_2$	(not_ite2 i)

Rule 66: connective_def This rule is used to replace connectives by their definition. It can be one of the following:

$$i.\ \triangleright\ \Gamma \qquad \varphi_1 \ \mathbf{xor}\ \varphi_2 \approx (\neg\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \neg\varphi_2) \qquad \text{connective\_def}$$

$$i.\ \triangleright\ \Gamma \qquad \varphi_1 \approx \varphi_2 \approx (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) \qquad \text{connective\_def}$$

$$i.\ \triangleright\ \Gamma \qquad \mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3 \approx (\varphi_1 \to \varphi_2) \wedge (\neg\varphi_1 \to \varphi_3) \qquad \text{connective\_def}$$

$$i.\ \triangleright\ \Gamma \qquad \forall x_1, \ldots, x_n.\ \varphi \approx \neg(\exists x_1, \ldots, x_n.\ \neg\varphi) \qquad \text{connective\_def}$$

Rule 67: and_simplify This rule simplifies an $\wedge$ term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad \varphi_1 \wedge \cdots \wedge \varphi_n \approx \psi \qquad \text{and\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $\top \wedge \cdots \wedge \top \Rightarrow \top$
- $\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_{n'}$ where the right-hand side has all $\top$ literals removed.
- $\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_{n'}$ where the right-hand side has all repeated literals removed.
- $\varphi_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge \varphi_n \Rightarrow \bot$
- $\varphi_1 \wedge \cdots \wedge \varphi_i \wedge \cdots \wedge \varphi_j \wedge \cdots \wedge \varphi_n \Rightarrow \bot$ and φ_i, φ_j are such that

$$\varphi_i = \underbrace{\neg \ldots \neg}_{n} \psi$$

$$\varphi_j = \underbrace{\neg \ldots \neg}_{m} \psi$$

and one of n, m is odd and the other even. Either can be 0.

Rule 68: or_simplify This rule simplifies an $\vee$ term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad (\varphi_1 \vee \cdots \vee \varphi_n) \approx \psi \qquad \text{or\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $\bot \vee \cdots \vee \bot \Rightarrow \bot$
- $\varphi_1 \vee \cdots \vee \varphi_n \Rightarrow \varphi_1 \vee \cdots \vee \varphi_{n'}$ where the right-hand side has all $\bot$ literals removed.
- $\varphi_1 \vee \cdots \vee \varphi_n \Rightarrow \varphi_1 \vee \cdots \vee \varphi_{n'}$ where the right-hand side has all repeated literals removed.
- $\varphi_1 \vee \cdots \vee \top \vee \cdots \vee \varphi_n \Rightarrow \top$
- $\varphi_1 \vee \cdots \vee \varphi_i \vee \cdots \vee \varphi_j \vee \cdots \vee \varphi_n \Rightarrow \top$ and φ_i, φ_j are such that

$$\varphi_i = \underbrace{\neg \ldots \neg}_{n} \psi$$

$$\varphi_j = \underbrace{\neg \ldots \neg}_{m} \psi$$

  and one of n, m is odd and the other even. Either can be 0.

Rule 69: not_simplify This rule simplifies an $\neg$ term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad \neg\varphi \approx \psi \qquad \text{not\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $\neg(\neg\varphi) \Rightarrow \varphi$
- $\neg\bot \Rightarrow \top$
- $\neg\top \Rightarrow \bot$

Rule 70: implies_simplify This rule simplifies an $\rightarrow$ term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad \varphi_1 \rightarrow \varphi_2 \approx \psi \qquad \text{implies\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $\neg\varphi_1 \rightarrow \neg\varphi_2 \Rightarrow \varphi_2 \rightarrow \varphi_1$
- $\bot \rightarrow \varphi \Rightarrow \top$
- $\varphi \rightarrow \top \Rightarrow \top$
- $\top \rightarrow \varphi \Rightarrow \varphi$
- $\varphi \rightarrow \bot \Rightarrow \neg\varphi$
- $\varphi \rightarrow \varphi \Rightarrow \top$

- $\neg\varphi \to \varphi \Rightarrow \varphi$
- $\varphi \to \neg\varphi \Rightarrow \neg\varphi$
- $(\varphi_1 \to \varphi_2) \to \varphi_2 \Rightarrow \varphi_1 \vee \varphi_2$

Rule 71: equiv_simplify This rule simplifies a formula with the head symbol $\approx$: **Bool Bool Bool** by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad (\varphi_1 \approx \varphi_2) \approx \psi \qquad \text{equiv\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $(\neg\varphi_1 \approx \neg\varphi_2) \Rightarrow (\varphi_1 \approx \varphi_2)$
- $(\varphi \approx \varphi) \Rightarrow \top$
- $(\varphi \approx \neg\varphi) \Rightarrow \bot$
- $(\neg\varphi \approx \varphi) \Rightarrow \bot$
- $(\top \approx \varphi) \Rightarrow \varphi$
- $(\varphi \approx \top) \Rightarrow \varphi$
- $(\bot \approx \varphi) \Rightarrow \neg\varphi$
- $(\varphi \approx \bot) \Rightarrow \neg\varphi$

Rule 72: bool_simplify This rule simplifies a boolean term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad \varphi \approx \psi \qquad \text{bool\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $\neg(\varphi_1 \to \varphi_2) \Rightarrow (\varphi_1 \wedge \neg\varphi_2)$
- $\neg(\varphi_1 \vee \varphi_2) \Rightarrow (\neg\varphi_1 \wedge \neg\varphi_2)$
- $\neg(\varphi_1 \wedge \varphi_2) \Rightarrow (\neg\varphi_1 \vee \neg\varphi_2)$
- $(\varphi_1 \to (\varphi_2 \to \varphi_3)) \Rightarrow (\varphi_1 \wedge \varphi_2) \to \varphi_3$
- $((\varphi_1 \to \varphi_2) \to \varphi_2) \Rightarrow (\varphi_1 \vee \varphi_2)$
- $(\varphi_1 \wedge (\varphi_1 \to \varphi_2)) \Rightarrow (\varphi_1 \wedge \varphi_2)$
- $((\varphi_1 \to \varphi_2) \wedge \varphi_1) \Rightarrow (\varphi_1 \wedge \varphi_2)$

Rule 73: ac_simp This rule simplifies nested occurrences of $\vee$ or $\wedge$:

$$\frac{}{i.\ \triangleright\ \ \Gamma \qquad \psi \approx \varphi_1 \circ \cdots \circ \varphi_n} \text{ ac\_simp}$$

where $\circ \in \{\vee, \wedge\}$ and ψ is a nested application of $\circ$. The literals φ_i are literals of the flattening of ψ with duplicates removed.

Rule 74: ite_simplify This rule simplifies an if-then-else term by applying equivalent transformations until fix point[36] It has the form

$$\frac{}{i.\ \triangleright\ \ \Gamma \qquad \mathbf{ite}\ \varphi\ t_1\ t_2 \approx u} \text{ ite\_simplify}$$

where u is the transformed term.

The possible transformations are:

- $\mathbf{ite}\ \top\ t_1\ t_2 \Rightarrow t_1$
- $\mathbf{ite}\ \bot\ t_1\ t_2 \Rightarrow t_2$
- $\mathbf{ite}\ \psi\ t\ t \Rightarrow t$
- $\mathbf{ite}\ \neg\varphi\ t_1\ t_2 \Rightarrow \mathbf{ite}\ \varphi\ t_2\ t_1$
- $\mathbf{ite}\ \psi\ (\mathbf{ite}\ \psi\ t_1\ t_2)\ t_3 \Rightarrow \mathbf{ite}\ \psi\ t_1\ t_3$
- $\mathbf{ite}\ \psi\ t_1\ (\mathbf{ite}\ \psi\ t_2\ t_3) \Rightarrow \mathbf{ite}\ \psi\ t_1\ t_3$
- $\mathbf{ite}\ \psi\ \top\ \bot \Rightarrow \psi$
- $\mathbf{ite}\ \psi\ \bot\ \top \Rightarrow \neg\psi$
- $\mathbf{ite}\ \psi\ \top\ \varphi \Rightarrow \psi \vee \varphi$
- $\mathbf{ite}\ \psi\ \varphi\ \bot \Rightarrow \psi \wedge \varphi$
- $\mathbf{ite}\ \psi\ \bot\ \varphi \Rightarrow \neg\psi \wedge \varphi$
- $\mathbf{ite}\ \psi\ \varphi\ \top \Rightarrow \neg\psi \vee \varphi$

Rule 75: qnt_simplify This rule simplifies a $\forall$-formula with a constant predicate.

$$\frac{}{i.\ \triangleright\ \ \Gamma \qquad \forall x_1, \ldots, x_n.\varphi \approx \varphi} \text{ qnt\_simplify}$$

where φ is either $\top$ or $\bot$.

[36] Note however that the order of the application is important, since the set of rules is not confluent. For example, the term $(\mathbf{ite}\top\ t_1\ t_2 \approx t_1)$ can be simplified into both p and $(\neg(\neg p))$ depending on the order of applications.

Rule 76: onepoint The onepoint rule is the "one-point-rule". That is: it eliminates quantified variables that can only have one value.

$$\begin{array}{lll|lll} & & & \vdots & & \\ j. & \Gamma, x_{k_1}, \ldots, x_{k_m}, x_{j_1} \mapsto t_{j_1}, \ldots, x_{j_o} \mapsto t_{j_o} & \triangleright & \varphi \approx \varphi' & (\ldots) \\ \hline k. & & \triangleright & Qx_1, \ldots, x_n.\varphi \approx Qx_{k_1}, \ldots, x_{k_m}.\varphi' & \text{onepoint} \end{array}$$

where $Q \in \{\forall, \exists\}$, $n = m + o$, $k_1, \ldots, k_m$ and $j_1, \ldots, j_o$ are monotone mappings to $1, \ldots, n$, and no x_{k_i} appears in $x_{j_1}, \ldots, x_{j_o}$.

The terms $t_{j_1}, \ldots, t_{j_o}$ are the points of the variables $x_{j_1}, \ldots, x_{j_o}$. Points are defined by equalities $x_i \approx t_i$ with positive polarity in the term φ.

Remark. Since an eliminated variable x_i might appear free in a term t_j, it is necessary to replace x_i with t_i inside t_j. While this substitution is performed correctly, the proof for it is currently missing.

Example 76.1
An application of the onepoint rule on the term $(\forall x, y.\, x \approx y \to (f\ x) \wedge (f\ y))$ look like this:

```
(anchor :step t3 :args ((:= y x)))
(step t3.t1 (cl (= x y)) :rule refl)
(step t3.t2 (cl (= (= x y) (= x x)))
    :rule cong :premises (t3.t1))
(step t3.t3 (cl (= x y)) :rule refl)
(step t3.t4 (cl (= (f y) (f x)))
    :rule cong :premises (t3.t3))
(step t3.t5 (cl (= (and (f x) (f y)) (and (f x) (f x))))
    :rule cong :premises (t3.t4))
(step t3.t6 (cl (= (=> (= x y) (and (f x) (f y)))
                   (=> (= x x) (and (f x) (f x)))))
    :rule cong :premises (t3.t2 t3.t5))
(step t3 (cl (=
        (forall ((x S) (y S)) (=> (= x y) (and (f x) (f y))))
        (forall ((x S))       (=> (= x x) (and (f x) (f x))))))
    :rule qnt_simplify)
```

Rule 77: qnt_join

$$i.\ \triangleright\ \Gamma \qquad Qx_1, \ldots, x_n.\ (Qx_{n+1}, \ldots, x_m.\ \varphi) \approx Qx_{k_1}, \ldots, x_{k_o}.\ \varphi \qquad \text{qnt\_join}$$

where $m > n$ and $Q \in \{\forall, \exists\}$. Furthermore, $k_1, \ldots, k_o$ is a monotonic map to $1, \ldots, m$ such that $x_{k_1}, \ldots, x_{k_o}$ are pairwise distinct, and $\{x_1, \ldots, x_m\} = \{x_{k_1}, \ldots, x_{k_o}\}$.

Rule 78: qnt_rm_unused

$$i.\ \triangleright\ \Gamma \qquad Qx_1, \ldots, x_n.\ \varphi \approx Qx_{k_1}, \ldots, x_{k_m}.\ \varphi \qquad \text{qnt\_rm\_unused}$$

where $m \leq n$ and $Q \in \{\forall, \exists\}$. Furthermore, $k_1, \ldots, k_m$ is a monotonic map to $1, \ldots, n$ and if $x \in \{x_j \mid j \in \{1, \ldots, n\} \wedge j \notin \{k_1, \ldots, k_m\}\}$ then x is not free in P.

Rule 79: eq_simplify This rule simplifies an $\approx$ term by applying equivalent transformations as long as possible. Hence, the general form is

$$i.\ \triangleright\ \Gamma \qquad (t_1 \approx t_2) \approx \varphi \qquad \text{eq\_simplify}$$

where ψ is the transformed term.

The possible transformations are:

- $t \approx t \Rightarrow \top$
- $(t_1 \approx t_2) \Rightarrow \bot$ if t_1 and t_2 are different numeric constants.
- $\neg(t \approx t) \Rightarrow \bot$ if t is a numeric constant.

Rule 80: div_simplify This rule simplifies a division by applying equivalent transformations. The general form is

$$i.\ \triangleright\ \Gamma \qquad (t_1 / t_2) \Rightarrow t_3 \qquad \text{div\_simplify}$$

The possible transformations are:

- $t / t \Rightarrow 1$
- $t / 1 \Rightarrow t$
- $t_1 / t_2 \Rightarrow t_3$ if t_1 and t_2 are constants and t_3 is t_1 divided by t_2 according to the semantic of the current theory.

Rule 81: prod_simplify This rule simplifies a product by applying equivalent transformations as long as possible. The general form is

$$i.\ \triangleright\ \Gamma \qquad t_1 \times \cdots \times t_n \approx u \qquad \text{prod\_simplify}$$

where u is either a constant or a product.

The possible transformations are:

- $t_1 \times \cdots \times t_n \Rightarrow u$ where all t_i are constants and u is their product.
- $t_1 \times \cdots \times t_n \Rightarrow 0$ if any t_i is 0.
- $t_1 \times \cdots \times t_n \Rightarrow c \times t_{k_1} \times \cdots \times t_{k_n}$ where c is the product of the constants of $t_1, \ldots, t_n$ and $t_{k_1}, \ldots, t_{k_n}$ is $t_1, \ldots, t_n$ with the constants removed.
- $t_1 \times \cdots \times t_n \Rightarrow t_{k_1} \times \cdots \times t_{k_n}$: same as above if c is 1.

Rule 82: unary_minus_simplify This rule is either

$$i.\ \triangleright\ \Gamma \qquad -(-t) \approx t \qquad \text{unary\_minus\_simplify}$$

or

$$i.\ \triangleright\ \Gamma \qquad -t \approx u \qquad \text{unary\_minus\_simplify}$$

where u is the negated numerical constant t.

Rule 83: minus_simplify This rule simplifies a subtraction by applying equivalent transformations. The general form is

$$i.\ \triangleright\ \Gamma \qquad t_1 - t_2 \approx u \qquad \text{minus\_simplify}$$

The possible transformations are:

- $t - t \Rightarrow 0$
- $t_1 - t_2 \Rightarrow t_3$ where t_1 and t_2 are numerical constants and t_3 is t_2 subtracted from t_1.
- $t - 0 \Rightarrow t$
- $0 - t \Rightarrow -t$

Rule 84: sum_simplify This rule simplifies a sum by applying equivalent transformations as long as possible. The general form is

$$i.\ \triangleright\ \Gamma \qquad t_1 + \cdots + t_n \approx u \qquad \text{sum\_simplify}$$

where u is either a constant or a product.

The possible transformations are:

- $t_1 + \cdots + t_n \Rightarrow c$ where all t_i are constants and c is their sum.
- $t_1 + \cdots + t_n \Rightarrow c + t_{k_1} + \cdots + t_{k_n}$ where c is the sum of the constants of $t_1, \ldots, t_n$ and $t_{k_1}, \ldots, t_{k_n}$ is $t_1, \ldots, t_n$ with the constants removed.
- $t_1 + \cdots + t_n \Rightarrow t_{k_1} + \cdots + t_{k_n}$: same as above if c is 0.

Rule 85: comp_simplify This rule simplifies a comparison by applying equivalent transformations as long as possible. The general form is

$$i.\ \triangleright\ \Gamma \qquad\qquad t_1 \bowtie t_n \approx \psi \qquad\qquad \text{comp\_simplify}$$

where $\bowtie$ is as for the proof rule la_generic, but never $\approx$.

The possible transformations are:

- $t_1 < t_2 \Rightarrow \varphi$ where t_1 and t_2 are numerical constants and φ is $\top$ if t_1 is strictly greater than t_2 and $\bot$ otherwise.
- $t < t \Rightarrow \bot$
- $t_1 \leq t_2 \Rightarrow \varphi$ where t_1 and t_2 are numerical constants and φ is $\top$ if t_1 is greater than t_2 or equal and $\bot$ otherwise.
- $t \leq t \Rightarrow \top$
- $t_1 \geq t_2 \Rightarrow t_2 \leq t_1$
- $t_1 < t_2 \Rightarrow \neg(t_2 \leq t_1)$
- $t_1 > t_2 \Rightarrow \neg(t_1 \leq t_2)$

Rule 86: let This rule eliminates **let**. It has the form

$$\begin{array}{lll}
i_1.\ \Gamma & \triangleright & t_1 \approx s_1 \qquad (\ldots) \\
 & & \vdots \\
i_n.\ \Gamma & \triangleright & t_n \approx s_n \qquad (\ldots) \\
 & & \vdots \\
j.\ \Gamma, x_1 \mapsto s_1, \ldots, x_n \mapsto s_n & \triangleright & u \approx u' \qquad (\ldots) \\
\hline
k.\ \Gamma & \triangleright & (\textbf{let}\ x_1 = t_1,\ \ldots,\ x_n = t_n\ \textbf{in}\ u) \approx u' \quad (\text{let } i_1, \ldots, i_n)
\end{array}$$

The premise $i_1, \ldots, i_n$ must be in the same subproof as the let step. If for $t_i \approx s_i$ the t_i and s_i are syntactically equal, the premise is skipped.

Rule 87: distinct_elim This rule eliminates the **distinct** predicate. If called with one argument this predicate always holds:

$$i.\ \triangleright\ \Gamma \qquad (\mathbf{distinct}\ t) \approx \top \qquad \text{distinct\_elim}$$

If applied to terms of type **Bool** more than two terms can never be distinct, hence only two cases are possible:

$$i.\ \triangleright\ \Gamma \qquad (\mathbf{distinct}\ \varphi\ \psi) \approx \neg(\varphi \approx \psi) \qquad \text{distinct\_elim}$$

and

$$i.\ \triangleright\ \Gamma \qquad (\mathbf{distinct}\ \varphi_1\ \varphi_2\ \varphi_3\ \ldots) \approx \bot \qquad \text{distinct\_elim}$$

The general case is

$$i.\ \triangleright\ \Gamma \qquad (\mathbf{distinct}\ t_1\ \ldots\ t_n) \approx \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} t_i \not\approx t_j \qquad \text{distinct\_elim}$$

Rule 88: la_rw_eq

$$i.\ \triangleright \qquad (t \approx u) \approx (t \leq u \wedge u \leq t) \qquad \text{la\_rw\_eq}$$

Remark. While the connective could be $\approx$, currently an equality is used.

Rule 89: nary_elim This rule replaces n-ary operators with their equivalent application of the binary operator. It is never applied to $\wedge$ or $\vee$.

Three cases are possible. If the operator $\circ$ is left associative, then the rule has the form

$$i.\ \triangleright\ \Gamma \qquad \bigcirc_{i=1}^{n} t_i \approx (\ldots(t_1 \circ t_2) \circ t_3) \circ \cdots t_n) \qquad \text{nary\_elim}$$

If the operator $\circ$ is right associative, then the rule has the form

$$i.\ \triangleright\ \Gamma \qquad \bigcirc_{i=1}^{n} t_i \approx (t_1 \circ \cdots \circ (t_{n-2} \circ (t_{n-1} \circ t_n)\ldots) \qquad \text{nary\_elim}$$

If the operator is *chainable*, then it has the form

$$i.\ \triangleright\ \Gamma \qquad \bigcirc_{i=1}^{n} t_i \approx (t_1 \circ t_2) \wedge (t_2 \circ t_3) \wedge \cdots \wedge (t_{n-1} \circ t_n) \qquad \text{nary\_elim}$$

Rule 90: bfun_elim

$$i.\ \triangleright \qquad \psi \qquad (\ldots)$$
$$j.\ \triangleright \qquad \varphi \qquad (\text{bfun\_elim}\ i)$$

The formula φ is ψ after boolean functions have been simplified. This happens in a two step process. Both steps recursively iterate over ψ. The

first step expands quantified variable of type **Bool**. Hence, $(\exists x.\, t)$ becomes $t[x \mapsto \bot] \vee t[x \mapsto \top]$ and $(\forall x.\, t)$ becomes $t[x \mapsto \bot] \wedge t[x \mapsto \top]$. If n variables of sort **Bool** appear in a quantifier, the disjunction (conjunction) has 2^n terms. Each term replaces the variables in t according to the bits of a number which is increased by one for each subsequent term starting from zero. The left-most variable corresponds to the least significant bit.

The second step expands function argument of boolean types by introducing appropriate if-then-else terms. For example, consider $(f\ x\ P\ y)$ where P is some formula. Then we replace this term by $(\mathbf{ite}\ P\ (f\ x\ \top\ y)\ (f\ x\ \bot\ y))$. If the argument is already the constant $\top$ or $\bot$, it is ignored.

Rule 91: ite_intro

$$i.\ \triangleright \qquad t \approx (t' \wedge u_1 \wedge \ldots \wedge u_n) \qquad \text{(ite\_intro)}$$

The term t (the formula φ) contains the **ite** operator. Let $s_1, \ldots, s_n$ be the terms starting with **ite**, i.e. $s_i := \mathbf{ite}\ \psi_i\ r_i\ r_i'$, then u_i has the form

$$\mathbf{ite}\ \psi_i\ (s_i \approx r_i)\ (s_i \approx r_i')$$

The term t' is equal to the term t up to the reordering of equalities where one argument is an **ite** term.

Remark. This rule stems from the introduction of fresh constants for if-then-else terms inside veriT. Internally s_i is a new constant symbol and the φ on the right side of the equality is φ with the if-then-else terms replaced by the constants. Those constants are unfolded during proof printing. Hence, the slightly strange form and the reordering of equalities.

Rule Index

A
ac_simp 184
and 176
and_neg 178
and_pos 178
and_simplify 181
assume 167

B
bfun_elim 189
bind 172
bool_simplify 183

C
comp_simplify 188
cong 174
connective_def 181
contraction 169

D
distinct_elim 189
div_simplify 186

E
equiv_neg1 179
equiv_neg2 179
equiv_pos1 179
equiv_pos2 179
equiv_simplify 183
equiv1 177
equiv2 178
eq_congruent 174
eq_congruent_pred 174
eq_reflexive 174
eq_simplify 186
eq_transitive 174

F
false 167
forall_inst 173

H
hole 167

I
implies 177
implies_neg1 179
implies_neg2 179
implies_pos 179
implies_simplify 182
ite_intro 190
ite_neg1 180
ite_neg2 180
ite_pos1 180
ite_pos2 180
ite_simplify 184
ite1 180
ite2 180

L
la_disequality 171
la_generic 169
la_rw_eq 189
la_tautology 172
la_totality 172
let 188
lia_generic 171

M
minus_simplify 187

N
nary_elim 189

not_and 176
not_equiv1 178
not_equiv2 178
not_implies1 177
not_implies2 177
not_ite1 180
not_ite2 180
not_not 167
not_or 176
not_simplify 182
not_xor1 177
not_xor2 177

O
onepoint 185
or 176
or_neg 178
or_pos 178
or_simplify 181

P
prod_simplify 187

Q
qnt_cnf 175
qnt_join 186
qnt_rm_unused 186
qnt_simplify 184

R
refl 173
resolution 168

S
sko_ex 173
sko_forall 173
subproof 169
sum_simplify 187

T
tautology 168
th_resolution 168
trans 174
true 167

U
unary_minus_simplify 187

X
xor_neg1 179
xor_neg2 179
xor_pos1 178
xor_pos2 179
xor1 176
xor2 177

Bibliography

[1] M. Abdulaziz and L.C. Paulson, An Isabelle/HOL formalisation of Green's Theorem, *Archive of Formal Proofs* (2018). (Formal proof development)

[2] M. Abdulaziz and L.C. Paulson, An Isabelle/HOL Formalisation of Green's Theorem, *Journal of Automated Reasoning* **63**(3), 763–786 (2019).

[3] A. Aguirre, *Towards a Provably Correct Encoding from F* to SMT*, Inria Internship Report (2016).

[4] Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura, Finding conflicting instances of quantified formulas in SMT, In FMCAD 14 (IEEE, 2014).

[5] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses, In J.P. Jouannaud and Z. Shao (Eds.) CPP 1 (Springer Berlin Heidelberg, 2011).

[6] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard, Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system, In TYPES: Types for Proofs and Programs (Novi SAd, Serbia, 2016).

[7] M. Baaz, U. Egly, A. Leitsch, J. Goubault-Larrecq, and D. Plaisted, Chapter 5 – Normal Form Transformations, In A. Robinson and A. Voronkov (Eds.) Handbook of Automated Reasoning, pp.. 273–333 (North-Holland, Amsterdam, 2001).

[8] M. Balunović, P. Bielik, and M. Vechev, Learning to Solve SMT Formulas, In NIPS 32 (Curran Associates Inc., 2018).

[9] H. Barbosa, Efficient Instantiation Techniques in SMT (Work In Progress) (CEUR-WS.org, 2016).

[10] H. Barbosa, *New techniques for instantiation and proof production in SMT solving. (Nouvelles techniques pour l'instanciation et la production des preuves dans SMT)*, (PhD thesis). University of Lorraine, Nancy, France (2017).

[11] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, Cvc5: A Versatile and Industrial-Strength SMT Solver, In D. Fisman and G. Rosu (Eds.) TACAS 28 (Springer International Publishing, Cham, 2022).

[12] H. Barbosa, J.C. Blanchette, M. Fleury, and P. Fontaine, Scalable Fine-Grained Proofs for Formula Processing, *Journal of Automated Reasoning* **64**(3), 485–510 (2019).

[13] H. Barbosa, P. Fontaine, and A. Reynolds, Congruence Closure with Free Variables, In A. Legay and T. Margaria (Eds.) TACAS 23 (Springer Berlin Heidelberg, 2017).

[14] H. Barbosa, J. Hoenicke, and H. Antti, *16th International Satisfiability Modulo Theories Competition (SMT-COMP 2021): Rules and Procedures*, https://smt-comp.github.io/2021/rules.pdf (2021). (Accessed: 2021-08-08)

[15] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri *et al.*, Flexible Proof Production in an Industrial-Strength SMT Solver, In J. Blanchette, L. Kovács, and D. Pattinson (Eds.) IJCAR 11 (Springer International Publishing, Cham, 2022).

[16] H. Barbosa, A. Reynolds, D.E. Ouraoui, C. Tinelli, and C.W. Barrett, Extending SMT Solvers to Higher-Order Logic, In P. Fontaine (Ed.) CADE 27 (Springer, 2019).

[17] C. Barrett, P. Fontaine, and C. Tinelli, *The SMT-LIB Standard: Version 2.6*, Technical report (Department of Computer Science, The University of Iowa, 2017). (Available at `www.SMT-LIB.org`)

[18] C.W. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli, Satisfiability Modulo Theories, In A. Biere, M. Heule, H. van Maaren, and T. Walsh (Eds.) Handbook of Satisfiability, Vol. 336, pp.. 1267–1329 (IOS Press, 2021).

[19] H. Becker, J.C. Blanchette, U. Waldmann, and D. Wand, Formalization of Knuth–Bendix Orders for Lambda-Free Higher-Order Terms, *Archive of Formal Proofs* (2016). (Formal proof development)

[20] F. Besson, P. Fontaine, and L. Théry, A Flexible Proof Format for SMT: A Proposal, In P. Fontaine and A. Stump (Eds.) PxTP 1 (2011).

[21] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020 (2020).

[22] J.C. Blanchette, S. Böhme, and L.C. Paulson, Extending Sledgehammer with SMT Solvers, In N. Bjørner and V. Sofronie-Stokkermans (Eds.) CADE 23 (Springer Berlin Heidelberg, 2011).

[23] J.C. Blanchette, S. Böhme, and L.C. Paulson, Extending Sledgehammer with SMT Solvers, *Journal of Automated Reasoning* **51**(1), 109–128 (2013).

[24] J.C. Blanchette, S. Böhme, M. Fleury, S.J. Smolka, and A. Steckermeier, Semi-intelligible Isar Proofs from Machine-Generated Proofs, *Journal of Automated Reasoning* **56**(2), 155–200 (2016).

[25] J. Blanchette, *Matryoshka*, Project Webpage (2022). (Accessed: 2022-07-14)

[26] S. Böhme, *Proving Theorems of Higher-Order Logic with SMT Solvers*, (PhD thesis). Technische Universität München (2012).

[27] S. Böhme and T. Nipkow, Sledgehammer: Judgement Day, In J. Giesl and R. Hähnle (Eds.) IJCAR 5 (Springer Berlin Heidelberg, 2010).

[28] S. Böhme and T. Weber, Fast LCF-Style Proof Reconstruction for Z3, In M. Kaufmann and L.C. Paulson (Eds.) ITP 1 (Springer Berlin Heidelberg, 2010).

[29] M.P. Bonacina, C. Lynch, and L. de Moura, On Deciding Satisfiability by Theorem Proving with Speculative Inferences, *Journal of Automated Reasoning* **47** 161-189 (2011).

[30] T. Bouton, D. Caminha B. De Oliveira, D. Déharbe, and P. Fontaine, GridTPT: a distributed platform for Theorem Prover Testing, In PAAR 2 (Edinburgh, United Kingdom, 2010).

[31] T. Bouton, D.C.B. de Oliveira, D. Déharbe, and P. Fontaine, VeriT: An Open, Trustable and Efficient SMT-solver, In R.A. Schmidt (Ed.) CADE 22 (Springer Berlin Heidelberg, 2009).

[32] G. Burel, G. Bury, R. Cauderlier, D. Delahaye, P. Halmagrand, and O. Hermant, First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice, *Journal of Automated Reasoning* **64**(6), 1001–1050 (2020).

[33] J. Christ, J. Hoenicke, and A. Nutz, SMTInterpol: An Interpolating SMT Solver, In A.F. Donaldson and D. Parker (Eds.) SPIN 19 (Springer, 2012).

[34] K. Claessen and N. Sorensson, New Techniques that Improve MACE-style Finite Model Finding, In MODEL (2003).

[35] G. Cornuéjols, M.A. Trick, and M.J. Saltzman, *A Tutorial on Integer Programming*, https://www.math.clemson.edu/~mjs/courses/mthsc.440/integer.pdf (1995). (Accessed: 2022-04-07)

[36] R. Defourné, Improving Automation for Higher-Order Proof Steps, In B. Konev and G. Reger (Eds.) FroCos 13 (Springer, Birmingham, United Kingdom, 2021).

[37] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin, SMT solvers for Rodin, In J. Derrick, J.A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene (Eds.) ABZ 2012 (Springer Berlin Heidelberg, 2012).

[38] D. Déharbe, P. Fontaine, and B. Woltzenlogel Paleo, Quantifier Inference Rules for SMT Proofs, In P. Fontaine and A. Stump (Eds.) PxTP 1 (2011).

[39] M. Desharnais, P. Vukmirović, J. Blanchette, and M. Wenzel, Seventeen Provers Under the Hammer, In J. Andronick and L. de Moura (Eds.) ITP 13 (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022).

[40] D. Detlefs, G. Nelson, and J.B. Saxe, Simplify: A Theorem Prover for Program Checking, *Journal of the ACM* **52**(3), 365–473 (2005).

[41] G. Dowek, Deduction modulo theory, *CoRR* **abs/1501.06523** (2015).

[42] P.J. Downey, R. Sethi, and R.E. Tarjan, Variations on the Common Subexpression Problem, *Journal of the ACM* **27**(4), 758–771 (1980).

[43] C. Dross, *Generic Decision Procedures for Axiomatic First-Order Theories*, (PhD thesis). Université Paris-Sud (2014).

[44] B. Dutertre and L. de Moura, *Integrating Simplex with DPLL(T)*, Technical report (SRI International, 2006).

[45] L. de Moura and N. Bjørner, Efficient E-Matching for SMT Solvers, In F. Pfenning (Ed.) CADE 21 (Springer Berlin Heidelberg, 2007).

[46] L. de Moura and N. Bjørner, Engineering DPLL(T) + Saturation, In A. Armando, P. Baumgartner, and G. Dowek (Eds.) IJCAR 4 (Springer Berlin Heidelberg, 2008).

[47] L. de Moura and N. Bjørner, Z3: An Efficient SMT Solver, In C.R. Ramakrishnan and J. Rehof (Eds.) TACAS 14 (Springer Berlin Heidelberg, 2008).

[48] L. de Moura and G.O. Passmore, *The Strategy Challenge in SMT Solving*, In M.P. Bonacina and M.E. Stickel (Eds.) *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, pp.. 15–44 (Springer, Berlin, Heidelberg, 2013).

[49] M. Eberl, Elementary Facts About the Distribution of Primes, *Archive of Formal Proofs* (2019). (Formal proof development)

[50] M. Eberl and L.C. Paulson, The Prime Number Theorem, *Archive of Formal Proofs* (2018). (Formal proof development)

[51] G. Ebner, J. Blanchette, and S. Tourret, A Unifying Splitting Framework, In A. Platzer and G. Sutcliffe (Eds.) CADE 28 (Springer International Publishing, Cham, 2021).

[52] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett, SMTCoq: A Plug-In for Integrating SMT Solvers into Coq, In R. Majumdar and V. Kunčak (Eds.) CAV 29 (Springer, 2017).

[53] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C.W. Barrett, SMTCoq: A Plug-In for Integrating SMT Solvers into Coq, In R. Majumdar and V. Kuncak (Eds.) CAV 29 (Springer International Publishing, 2017).

[54] M. Fleury and H.J. Schurr, Reconstructing veriT Proofs in Isabelle/HOL, In G. Reis and H. Barbosa (Eds.) PxTP 6 (2019).

[55] P. Fontaine and H.J. Schurr, Quantifier Simplification by Unification in SMT, In B. Konev and G. Reger (Eds.) FroCoS 13 (Springer International Publishing, Cham, 2021).

[56] Y. Ge, C. Barrett, and C. Tinelli, Solving Quantified Verification Conditions Using Satisfiability Modulo Theories, In F. Pfenning (Ed.) CADE 21 (Springer Berlin Heidelberg, 2007).

[57] Y. Ge and L. de Moura, Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories, In A. Bouajjani and O. Maler (Eds.) CAV 21 (Springer Berlin Heidelberg, 2009).

[58] M.J.C. Gordon, R. Milner, and C.P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Vol. 78 (Springer Berlin Heidelberg, 1979).

[59] M. Heule, M. Järvisalo, and M. Suda, *SAT Race 2019*, http://sat-race-2019.ciirc.cvut.cz/ (2019). (Accessed: 2022-02-28)

[60] J. Hoenicke and T. Schindler, A Simple Proof Format for SMT, In D. Déharbe and A.E.J. Hyvärinen (Eds.) SMT 20 (CEUR-WS.org, 2022).

[61] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stützle, ParamILS: An Automatic Algorithm Configuration Framework, *Journal of Artificial Intelligence Research* **36**(1), 267–306 (2009).

[62] A.E.J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, OpenSMT2: An SMT Solver for Multi-core and Cloud Computing, In N. Creignou and D. Le Berre (Eds.) SAT 19 (Springer International Publishing, Cham, 2016).

[63] F. Immler, *Re: [isabelle] Isabelle2019-RC2 sporadic smt failures*, Email (2019).

[64] M. Janota, H. Barbosa, P. Fontaine, and A. Reynolds, Fair and Adventurous Enumeration of Quantifier Instantiations, In R. Piskac and M.W. Whalen (Eds.) FMCAD 21 (TU Wien Academic Press, Vienna, Austria, 2021).

[65] L. Kovács and A. Voronkov, First-Order Theorem Proving and Vampire, In N. Sharygina and H. Veith (Eds.) CAV 25 (Springer Berlin Heidelberg, 2013).

[66] D. Kühlwein, J.C. Blanchette, C. Kaliszyk, and J. Urban, MaSh: Machine Learning for Sledgehammer, In ITP 4 (Springer, 2013).

[67] D. Kühlwein and J. Urban, MaLeS: A Framework for Automatic Tuning of Automated Theorem Provers, *Journal of Automated Reasoning* **55**(2), 91-–116 (2015).

[68] K.R.M. Leino and C. Pit-Claudel, Trigger Selection Strategies to Stabilize Program Verifiers, In S. Chaudhuri and A. Farzan (Eds.) Computer Aided Verification (Springer International Publishing, Cham, 2016).

[69] M. Manzano, *Introduction to Many-Sorted Logic*, In *Many-Sorted Logic and Its Applications*, pp.. 3–86 (John Wiley & Sons, Inc., USA, 1993).

[70] F. Marić, M. Spasić, and R. Thiemann, An Incremental Simplex Algorithm with Unsatisfiable Core Generation, *Archive of Formal Proofs* (2018). (Formal proof development)

[71] S. McLaughlin, C. Barrett, and Y. Ge, Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite, *Electronic Notes in Theoretical Computer Science* **144**(2), 43–51 (2006).

[72] K.L. McMillan, Interpolants from Z3 proofs, In FMCAD 11 (FMCAD Inc, Austin, Texas, 2011).

[73] J. Meng and L.C. Paulson, Lightweight relevance filtering for machine-generated resolution problems, *Journal of Applied Logic* **7**(1), 41–57 (2009).

[74] S. Mitchell, M. O'Sullivan, and I. Dunning, PuLP : A Linear Programming Toolkit for Python (2011).

[75] G. Nelson and D.C. Oppen, Simplification by Cooperating Decision Procedures, *ACM Transactions on Programming Languages and Systems* **1**(2), 245–257 (1979).

[76] G. Nelson and D.C. Oppen, Fast Decision Procedures Based on Congruence Closure, *Journal of the ACM* **27**(2), 356–364 (1980).

[77] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T), *Journal of the ACM* **53**(6), 937–977 (2006).

[78] R. Otoni, M. Blicha, P. Eugster, A.E.J. Hyvärinen, and N. Sharygina, Theory-Specific Proof Steps Witnessing Correctness of SMT Executions, In DAC 58, IEEE (IEEE, San Francisco, CA, USA, 2021).

[79] A. Reynolds, Conflicts, Models and Heuristics for Quantifier Instantiation in SMT, In L. Kovacs and A. Voronkov (Eds.) Vampire 3 (EasyChair, 2017).

[80] A. Reynolds, H. Barbosa, and P. Fontaine, Revisiting Enumerative Instantiation, In D. Beyer and M. Huisman (Eds.) TACAS 24 (Springer International Publishing, 2018).

[81] A. Reynolds, C. Tinelli, D. Jovanovic, and C.W. Barrett, Designing Theory Solvers with Extensions, In C. Dixon and M. Finger (Eds.) FroCoS 11 (Springer, 2017).

[82] A. Riazanov and A. Voronkov, Limited Resource Strategy in Resolution Theorem Proving, *Journal of Symbolic Computation* **36**(1–2), 101–115 (2003).

[83] J.R. Rice, The Algorithm Selection Problem, Vol. 15, pp.. 65–118 (Elsevier, 1976).

[84] J.A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM* **12**(1), 23–41 (1965).

[85] A. Schlichtkrull, J.C. Blanchette, D. Traytel, and U. Waldmann, Formalization of Bachmair and Ganzinger's Ordered Resolution Prover, *Archive of Formal Proofs* (2018). (Formal proof development)

[86] S. Schulz, E – a brainiac theorem prover, *AI Communications* **15**(2-3), 111–126 (2002).

[87] H.J. Schurr, Optimal Strategy Schedules for Everyone, In B. Konev, C. Schon, and A. Steen (Eds.) PAAR 8 (CEUR-WS.org, 2022).

[88] H.J. Schurr, M. Fleury, and M. Desharnais, Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant, In A. Platzer and G. Sutcliffe (Eds.) CADE 28 (Springer International Publishing, Cham, 2021).

[89] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers, In J.F. Groote and K.G. Larsen (Eds.) TACAS 27 (Springer International Publishing, Cham, 2021).

[90] R. Sekar, I.V. Ramakrishnan, and A. Voronkov, *Term Indexing*, In *Handbook of Automated Reasoning*, pp.. 1853–1964 (Elsevier Science Publishers B. V., Amsterdam, Netherlands, 2001).

[91] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, SMT Proof Checking Using a Logical Framework, *Formal Methods in System Design* **42**(1), 91–118 (2013).

[92] T. Tammet, Gandalf, *Journal of Automated Reasoning* **18**(2), 199–204 (1997).

[93] T. Tammet, *Gandalf c-1.1*, http://www.tptp.org/CASC/15/SystemDescriptions.html#Gandalf (1998). (Accessed: 2022-03-24)

[94] O. Tange, *GNU Parallel 20210722 ('Blue Unity')*, 10.5281/zenodo.5123056 (2021). (GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.)

[95] J. Urban, BliStr: The Blind Strategymaker, In G. Gottlob, G. Sutcliffe, and A. Voronkov (Eds.) GCAI 2015 (EasyChair, 2015).

[96] A. Voronkov, AVATAR: The Architecture for First-Order Theorem Provers, In A. Biere and R. Bloem (Eds.) CAV 26 (Springer International Publishing, 2014).

[97] T. Weber, *Par4*, http://smt2019.galois.com/papers/tool_paper_9.pdf (2018). (SMT-COMP 2018 system description. Accessed: 2022-03-30)

[98] M. Wenzel, Shared-Memory Multiprocessing for Interactive Theorem Proving, In S. Blazy, C. Paulin-Mohring, and D. Pichardie (Eds.) ITP 4 (Springer Berlin Heidelberg, 2013).

[99] Wikipedia contributors, *Alethe (bird) — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Alethe_(bird)&oldid=1064503766 (2022). (Online; accessed 2022-09-02)

[100] A. Wolf and R. Letz, Strategy Parallelism in Automated Theorem Proving, In Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference (AAAI Press, 1998).

[101] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown, SATzilla: Portfolio-Based Algorithm Selection for SAT, *Journal of Artificial Intelligence Research* **32**(1), 565–606 (2008).

[102] Zenodo, *Reliable Reconstruction of Fine-Grained Proofs in a Proof Assistant* (Zenodo, 2021).

[103] Zenodo, *Quantifier Simplification by Unification in SMT* (Zenodo, 2021).

Index

A
Alethe 41
abstraction
  lambda 44, 56, 66, 75
  propositional 28, 96
algorithm selection 139
anchor 48
assertion 66, 95
assumption 42
atom 24
axiom 19

B
Boolean 13, 69, 99
binder 13, 39, 55, 88
bound variable 25
box 98

C
CCFV 34
Coq 89
Core theory 20
CDCL 26
CDCL(T) 26
choice 46
clause normal form 24
context 42, 43, 48, 55, 67
  calculated 53
cross-validation 133
CVC4 113

D
Dedukti 89
decimal 21, 72
decision level 31
discrimination tree 105
domain 16

E
E-matching 35
encoding 75

F
fact filter 8
formula 16
  ground 16
  trimmed 24
free variable 25
function symbol 13
  ranked 14

G
Gandalf 139
ground 16, 29

H
HOL4 89
higher-order logic 9, 65, 148

I
Isar 91
instantiation 30, 33
  conflict-driven 34, 98
  enumerative 36, 99
  model-based 37, 99
  trigger-based 35, 98, 148
integer programming 119
  mixed 119
interpretation 17

J
jitter 131

L
LCF 65
Logipedia 89
lambda calculus 55
lemma 43
linear arithmetic 21
literal 24
logic 19

M
MachSMT 144
MaSh 79
Matryoshka 148
MePo 79
Mirabelle 77
matching loop 36
matrix 25
metaterms 55
model 18
  ground 29
  propositional 29
monomial 21

N
Nelson-Oppen method 19
numeral 21, 72

O
OpenSMT 88
objective function 119
occurs check 107
overloading 14

P
Pure 65
polarity 25
polymorphism 14, 65
preplay 8, 64
preprocessing 26, 96
pre-schedule 128
pre-simplified 100
proof 42
  valid 55
  well-formed 54
proof assistant 7
p-SETHEO 140

Q
quantifier
  strong 25
  weak 25
quantifier instantiation see instantiation
quantifier-free 20

R
rank 14
rewrite rule 148
rule
  concluding 48

S
SATzilla 144
Skolemization 27
SMT 7
SMTCoq 89
SMTInterpol 88
SMT-LIB
  formula 16
  logic 12
  sort 13
  term 13
  theory 19

satisfiable 18
SUB rule 100
schedgen 125
schedule 118, 121
  optimal 120
  order 121
  simulation 131
sentence 16
signature 14
skolemization 74
  operator 25
sort 13, 14
sort symbol 13
split 133
step 42
  outermost 55
step skipping 76
strategy 118
strategy selection 139
strong quantifier 25
subproof 43
  first-innermost 52
  valid 53
subproofs 68
substitution 23
superposition 97
symbol
  function 13
  sort 13

T
tactic 8, 64, 143
term 13
  ground 16
  lambda 55
  meta 55
  well-sorted 14, 15
theory 19
  Core 20
  linear arithmetic 21
theory combination 19
trace 31
trigger 35, 148
trim 24

U
unification 105
unit-box 98
unsatisfiable 18

V
Vampire 113, 140
variable
  binary 120
  bound 25
  free 16, 25
  sorted 14
virtual best solver 110, 124

W
weak quantifier 25
well-formed 52

Colophon

Written in Nancy and Barjac, Berlin, Lauterach, Liège

Chapter heads use “Rot 102R”, a typeface by Adam Rot of Metz. The typeface was first used in 1471 and digitized in 2016 by Rafael Ribas & Alexis Faudot of ANRT Nancy and their students. The character ‘é’ was added by the author.

The body text uses “Adobe Source Serif” by Frank Grießhammer and “Adobe Source Sans” by Paul D. Hunt.

The monospace font is “Iosevka” by Renzhi Li. Mathematics use the fonts of the “xits” project. To accommodate other scripts the fonts “Gowun Batang”, “Mirza”, and “Noto Serif” are used.

Typeset using ConTEXt.

The title image is a drawing of the Nasmyth steam hammer taken from The Popular Science Monthly, January 1891.

Experiments presented in this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr).

This thesis was funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreements No. 713999, Matryoshka).