

Christoph Benz Müller
Marijn J. H. Heule
Renate A. Schmidt (Eds.)

LNAI 14739

Automated Reasoning

12th International Joint Conference, IJCAR 2024
Nancy, France, July 3–6, 2024
Proceedings, Part I

1
Part I

ijcar
2024

 Springer

OPEN ACCESS

Lecture Notes in Computer Science

Lecture Notes in Artificial Intelligence

14739

Founding Editor

Jörg Siekmann

Series Editors

Randy Goebel, *University of Alberta, Edmonton, Canada*

Wolfgang Wahlster, *DFKI, Berlin, Germany*

Zhi-Hua Zhou, *Nanjing University, Nanjing, China*

The series Lecture Notes in Artificial Intelligence (LNAI) was established in 1988 as a topical subseries of LNCS devoted to artificial intelligence.

The series publishes state-of-the-art research results at a high level. As with the LNCS mother series, the mission of the series is to serve the international R & D community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings.


Christoph Benzmüller · Marijn J. H. Heule ·
Renate A. Schmidt
Editors

Automated Reasoning

12th International Joint Conference, IJCAR 2024
Nancy, France, July 3–6, 2024
Proceedings, Part I

Editors

Christoph Benz Müller 
Otto-Friedrich-Universität Bamberg
Bamberg, Germany

Marijn J. H. Heule 
Carnegie Mellon University
Pittsburgh, PA, USA

Renate A. Schmidt
University of Manchester
Manchester, UK



ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Artificial Intelligence
ISBN 978-3-031-63497-0 ISBN 978-3-031-63498-7 (eBook)
<https://doi.org/10.1007/978-3-031-63498-7>

LNCS Sublibrary: SL7 – Artificial Intelligence

© The Editor(s) (if applicable) and The Author(s) 2024. This book is an open access publication.

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

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

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

Preface

This volume contains the papers of the 12th International Joint Conference on Automated Reasoning (IJCAR) held in Nancy, France, during July 3–6, 2024. IJCAR is the premier international joint conference on all aspects of automated reasoning, including foundations, implementations, and applications, comprising several leading conferences and workshops. IJCAR 2024 brought together the Conference on Automated Deduction (CADE), the International Symposium on Frontiers of Combining Systems (FroCoS), and the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX).

Previous IJCAR conferences were held in Siena, Italy (2001), Cork, Ireland (2004), Seattle, USA (2006), Sydney, Australia (2008), Edinburgh, UK (2010), Manchester, UK (2012), Vienna, Austria (2014), Coimbra, Portugal (2016), Oxford, UK (2018), Paris, France (2020, virtual), and Haifa, Israel (2022).

IJCAR 2024 received 115 submissions (130 abstracts) out of which 45 papers were accepted (with an overall acceptance rate of 39%): 39 regular papers (out of 96 regular papers submitted, resulting in a regular paper acceptance rate of 41%) and 6 short papers (out of 19 short papers submitted, resulting in a short paper acceptance rate of 31%). Each submission was assigned to at least three Program Committee members and was reviewed in single-blind mode. All submissions were evaluated according to the following criteria: relevance, originality, significance, correctness, and readability. The review process included a feedback/rebuttal period, during which authors had the option to respond to reviewer comments.

In addition to the accepted papers, the IJCAR 2024 program included three invited talks:

- Jeremy Avigad (Carnegie Mellon University, USA) on “Automated Reasoning for Mathematics”,
- Laura Kovács (TU Wien, Austria) on “Induction in Saturation”, and
- Geoff Sutcliffe (University of Miami, USA) on “Stepping Stones in the TPTP World”.

This year marks the 30th anniversary of the CADE ATP System Competition (CASC), which was conceived in 1994 after CADE-12 in Nancy, France, when Christian Suttner and Geoff Sutcliffe were sitting on a bench under a tree in Parc de la Pépinière. In the 28 competitions since then, CASC has been a catalyst for research and development, providing an inspiring environment for personal interaction between ATP researchers and users. A special event took place to celebrate this anniversary.

In addition to the main programme, IJCAR 2024 hosted ten workshops, which took place on July 1–2, and two systems competitions (CASC and Termination). The SAT/SMT/AR 2024 Summer School was held in Nancy the week prior to IJCAR 2024.

The Best Paper Award of IJCAR 2024 went to Hugo Férée, Iris van der Giessen, Sam van Gool, and Ian Shillito for the paper “Mechanised Uniform Interpolation for

Modal Logics K, GL, and iSL”. The Best Student Paper Award went to Johannes Niederhauser (with Chad E. Brown and Cezary Kaliszyk) for the paper entitled “Tableaux for Automated Reasoning in Dependently-Typed Higher-Order Logic”.

Another highlight of the conference was the presentation of the 2024 Herbrand Award for Distinguished Contributions to Automated Reasoning to Armin Biere (Albert-Ludwigs-University Freiburg, Germany) in recognition of “his outstanding contributions to satisfiability solving, including innovative applications, methods for formula pre- and in-processing and proof generation, and a series of award-winning solvers, with deep impact on model checking and verification.”

The 2024 Bill McCune PhD Award was given to Katherine Kosaian for the PhD thesis “Formally Verifying Algorithms for Real Quantifier Elimination”, completed at Carnegie Mellon University, USA, in 2023.

The main institutions supporting IJCAR 2024 were the University of Lorraine and the Inria research center at the University of Lorraine. We also thank as sponsors: the research laboratory for computer science in Nancy (LORIA), a joint research unit of the University of Lorraine, CNRS, and Inria, its Formal Methods Department, and Métropole du Grand Nancy. For hosting the conference, we thank IDMC Nancy.

We would also like to acknowledge the generous sponsorship of Springer and Imandra Inc., and the support by EasyChair. Finally, we are indebted to the entire IJCAR 2024 Organizing Team for their assistance with the local organization and general management of the conference, especially Didier Galmiche, Stephan Merz, Christophe Ringeissen (Conference Co-Chairs), Sophie Touret (Workshop, Tutorial and Competition Chair), Peter Lammich (Publicity Chair) and Anne-Lise Charbonnier and Sabrina Lemaire (main administrative support).

May 2024

Christoph Benz Müller
Marijn J. H. Heule
Renate A. Schmidt

Organization

Conference Chairs

Didier Galmiche	University of Lorraine, France
Stephan Merz	Inria, University of Lorraine, France
Christophe Ringeissen	Inria, University of Lorraine, France

Program Committee Chairs

Christoph Benz Müller	Otto-Friedrich-Universität Bamberg and FU Berlin, Germany
Marijn J. H. Heule	Carnegie Mellon University, USA
Renate A. Schmidt	University of Manchester, UK

Workshop, Tutorial and Competition Chair

Sophie Touret	Inria, France and Max Planck Institute for Informatics, Germany
---------------	--

Publicity Chair

Peter Lammich	University of Twente, The Netherlands
---------------	---------------------------------------

Local Arrangements

Anne-Lise Charbonnier	Inria, France
Sabrina Lemaire	Inria, France

Steering Committee

Arnon Avron	Tel-Aviv University, Israel
Franz Baader	TU Dresden, Germany
Jürgen Giesl	RWTH Aachen University, Germany
Marijn J. H. Heule	Carnegie Mellon University, USA

Lawrence Paulson	University of Cambridge, UK
Elaine Pimentel	University College London, UK
Christophe Ringeissen	Inria, University of Lorraine, France
Renate A. Schmidt	University of Manchester, UK

Program Committee

Franz Baader	TU Dresden, Germany
Haniel Barbosa	Universidade Federal de Minas Gerais, Brazil
Christoph Benz Müller	Otto-Friedrich-Universität Bamberg and FU Berlin, Germany
Armin Biere	University of Freiburg, Germany
Nikolaj Bjørner	Microsoft, USA
Jasmin Blanchette	Ludwig-Maximilians-Universität München, Germany
Maria Paola Bonacina	Università degli Studi di Verona, Italy
Florent Capelli	Université d'Artois, France
Agata Ciabattone	TU Wien, Austria
Clare Dixon	University of Manchester, UK
Pascal Fontaine	Université de Liège, Belgium
Carsten Fuhs	Birkbeck, University of London, UK
Didier Galmiche	University of Lorraine, France
Silvio Ghilardi	Università degli Studi di Milano, Italy
Jürgen Giesl	RWTH Aachen University, Germany
Arie Gurfinkel	University of Waterloo, Canada
Marijn J. H. Heule	Carnegie Mellon University, USA
Andrzej Indrzejczak	University of Lodz, Poland
Moa Johansson	Chalmers University of Technology, Sweden
Daniela Kaufmann	TU Wien, Austria
Patrick Koopmann	Vrije Universiteit Amsterdam, The Netherlands
Konstantin Korovin	University of Manchester, UK
Peter Lammich	University of Twente, The Netherlands
Martin Lange	University of Kassel, Germany
Tim Lyon	Technische Universität Dresden, Germany
Kuldeep S. Meel	University of Toronto, Canada
Stephan Merz	Inria, University of Lorraine, France
Cláudia Nalon	University of Brasília, Brazil
Aina Niemetz	Stanford University, USA
Albert Oliveras	Universitat Politècnica de Catalunya, Spain
Xavier Parent	TU Wien, Austria
Nicolas Peltier	CNRS, Laboratory of Informatics of Grenoble, France

Rafael Peñaloza	University of Milano-Bicocca, Italy
Elaine Pimentel	University College London, UK
André Platzer	Karlsruhe Institute of Technology, Germany
Andrei Popescu	University of Sheffield, UK
Florian Rabe	FAU Erlangen-Nürnberg, Germany
Giles Reger	Amazon Web Services, USA and University of Manchester, UK
Giselle Reis	Carnegie Mellon University, Qatar
Andrew Reynolds	University of Iowa, USA
Christophe Ringeissen	Inria, University of Lorraine, France
Philipp Rümmer	University of Regensburg, Germany
Uli Sattler	University of Manchester, UK
Tanja Schindler	University of Basel, Switzerland
Renate A. Schmidt	University of Manchester, UK
Claudia Schon	Hochschule Trier, Germany
Stephan Schulz	DHBW Stuttgart, Germany
Roberto Sebastiani	University of Trento, Italy
Martina Seidl	Johannes Kepler University Linz, Austria
Viorica Sofronie-Stokkermans	University of Koblenz, Germany
Alexander Steen	University of Greifswald, Germany
Martin Suda	Czech Technical University in Prague, Czech Republic
Yong Kiam Tan	Institute for Infocomm Research, A*STAR, Singapore
Sophie Tournet	Inria, France and Max Planck Institute for Informatics, Germany
Josef Urban	Czech Technical University in Prague, Czech Republic
Uwe Waldmann	Max Planck Institute for Informatics, Germany
Christoph Weidenbach	Max Planck Institute for Informatics, Germany
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Yoni Zohar	Bar-Ilan University, Israel

Additional Reviewers

Noah Abou El Wafa	Marvin Brieger
Takahito Aoto	Martin Bromberger
Martin Avanzini	James Brotherston
Philippe Balbiani	Chad E. Brown
Lasse Blaauwbroek	Florian Bruse
Frédéric Blanqui	Filip Bártek
Thierry Boy de La Tour	Julie Cailler

Cameron Calk
Christophe Chareton
Jiaoyan Chen
Karel Chvalovský
Tiziano Dalmonte
Anupam Das
Martin Desharnais
Paulius Dilkas
Marie Duflot
Yotam Dvir
Chelsea Edmonds
Sólrun Halla Einarsdóttir
Clemens Eisenhofer
Zafer Esen
Camillo Fiorentini
Mathias Fleury
Stef Frijters
Florian Frohn
Nikolaos Galatos
Alessandro Gianola
Matt Griffin
Alberto Griggio
Liye Guo
Raúl Gutiérrez
Xavier Génereux
Hans-Dieter Hiep
Jochen Hoenicke
Jonathan Huerta y Munive
Ullrich Hustadt
Cezary Kaliszyk
Jan-Christoph Kassing
Michael Kinyon
Lydia Kondylidou
Boris Konev
George Kourtis
Francesco Kriegel
Falko Kötter
Timo Lang
Jonathan Laurent
Daniel Le Berre
Jannis Limperg
Xinghan Liu

Anela Lolic
Etienne Lozes
Salvador Lucas
Andreas Lööw
Kenji Maillard
Sérgio Marcelino
Andrew M. Marshall
Gabriele Masina
Marcel Moosbrugger
Barbara Morawska
Johannes Oetsch
Eugenio Orlandelli
Jens Otten
Adam Pease
Bartosz Piotrowski
Enguerrand Prebet
Siddharth Priya
Long Qian
Jakob Rath
Colin Rothgang
Reuben Rowe
Jan Frederik Schaefer
Johannes Schoisswohl
Marcel Schütz
Florian Sextl
Ian Shillito
Nicholas Smallbone
Giuseppe Spallitta
Sergei Stepanenko
Georg Struth
Matteo Tesi
Guilherme Toledo
Patrick Trentin
Hari Govind Vediramana Krishnan
Laurent Vigneron
Renaud Vilmart
Dominik Wehr
Tobias Winkler
Frank Wolter
Akihisa Yamada
Michal Zawadzki

Contents – Part I

Invited Contributions

Automated Reasoning for Mathematics	3
<i>Jeremy Avigad</i>	
Induction in Saturation	21
<i>Laura Kovács, Petra Hozzová, Márton Hajdu, and Andrei Voronkov</i>	
Stepping Stones in the TPTP World	30
<i>Geoff Sutcliffe</i>	

Theorem Proving and Tools

An Empirical Assessment of Progress in Automated Theorem Proving	53
<i>Geoff Sutcliffe, Christian Suttner, Lars Kotthoff, C. Raymond Perrault, and Zain Khalid</i>	
A Higher-Order Vampire (Short Paper)	75
<i>Ahmed Bhayat and Martin Suda</i>	
Tableaux for Automated Reasoning in Dependently-Typed Higher-Order Logic	86
<i>Johannes Niederhauser, Chad E. Brown, and Cezary Kaliszyk</i>	
The Naproche-ZF Theorem Prover (Short Paper)	105
<i>Adrian De Lon</i>	
Reducibility Constraints in Superposition	115
<i>Márton Hajdu, Laura Kovács, Michael Rawson, and Andrei Voronkov</i>	
First-Order Automatic Literal Model Generation	133
<i>Martin Bromberger, Florent Krasnopol, Sibylle Möhle, and Christoph Weidenbach</i>	
Synthesis of Recursive Programs in Saturation	154
<i>Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner</i>	

Synthesizing Strongly Equivalent Logic Programs: Beth Definability for Answer Set Programs via Craig Interpolation in First-Order Logic	172
<i>Jan Heuer and Christoph Wernhard</i>	
Regularization in Spider-Style Strategy Discovery and Schedule Construction	194
<i>Filip Bártek, Karel Chvalovský, and Martin Suda</i>	
Lemma Discovery and Strategies for Automated Induction	214
<i>Sólrun Halla Einarsdóttir, Márton Hajdu, Moa Johansson, Nicholas Smallbone, and Martin Suda</i>	
Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper)	233
<i>Nils Lommen, Éléanore Meyer, and Jürgen Giesl</i>	
On the (In-)Completeness of Destructive Equality Resolution in the Superposition Calculus	244
<i>Uwe Waldmann</i>	
SAT, SMT and Quantifier Elimination	
Model Completeness for Rational Trees	265
<i>Silvio Ghilardi and Lia M. Poidomani</i>	
Certifying Phase Abstraction	284
<i>Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko</i>	
Verifying a Realistic Mutable Hash Table: Case Study (Short Paper)	304
<i>Samuel Chassot and Viktor Kunčák</i>	
Booleguru, the Propositional Polyglot (Short Paper)	315
<i>Maximilian Heisinger, Simone Heisinger, and Martina Seidl</i>	
Quantifier Shifting for Quantified Boolean Formulas Revisited	325
<i>Simone Heisinger, Maximilian Heisinger, Adrian Rebola-Pardo, and Martina Seidl</i>	
Satisfiability Modulo Exponential Integer Arithmetic	344
<i>Florian Frohn and Jürgen Giesl</i>	
SAT-Based Learning of Computation Tree Logic	366
<i>Adrien Pommellet, Daniel Stan, and Simon Scatton</i>	

<p>MCSat-Based Finite Field Reasoning in the YICES2 SMT Solver (Short Paper)</p> <p style="padding-left: 2em;"><i>Thomas Hader, Daniela Kaufmann, Ahmed Irfan, Stéphane Graham-Lengrand, and Laura Kovács</i></p>	<p>386</p>
<p>Certified MaxSAT Preprocessing</p> <p style="padding-left: 2em;"><i>Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström</i></p>	<p>396</p>
<p>A Formal Model to Prove Instantiation Termination for E-matching-Based Axiomatisations</p> <p style="padding-left: 2em;"><i>Rui Ge, Ronald Garcia, and Alexander J. Summers</i></p>	<p>419</p>
<p>Fast and Verified UNSAT Certificate Checking</p> <p style="padding-left: 2em;"><i>Peter Lammich</i></p>	<p>439</p>
<p>Generalized Optimization Modulo Theories</p> <p style="padding-left: 2em;"><i>Nestan Tsiskaridze, Clark Barrett, and Cesare Tinelli</i></p>	<p>458</p>
<p>Author Index</p>	<p>481</p>

Contents – Part II

Intuitionistic Logics and Modal Logics

Model Construction for Modal Clauses	3
<i>Ullrich Hustadt, Fabio Papacchini, Cláudia Nalon, and Clare Dixon</i>	
A Terminating Sequent Calculus for Intuitionistic Strong Löb Logic with the Subformula Property	24
<i>Camillo Fiorentini and Mauro Ferrari</i>	
Mechanised Uniform Interpolation for Modal Logics K, GL, and iSL	43
<i>Hugo Féréé, Iris van der Giessen, Sam van Gool, and Ian Shillito</i>	
Skolemisation for Intuitionistic Linear Logic	61
<i>Alessandro Bruni, Eike Ritter, and Carsten Schürmann</i>	
Local Intuitionistic Modal Logics and Their Calculi	78
<i>Philippe Balbiani, Han Gao, Çiğdem Gencer, and Nicola Olivetti</i>	
Non-iterative Modal Resolution Calculi	97
<i>Dirk Pattinson and Cláudia Nalon</i>	
A Logic for Repair and State Recovery in Byzantine Fault-Tolerant Multi-agent Systems	114
<i>Hans van Ditmarsch, Krisztina Fruzsa, Roman Kuznets, and Ulrich Schmid</i>	

Calculi, Proof Theory and Decision Procedures

A Decision Method for First-Order Stream Logic	137
<i>Harald Ruess</i>	
What Is Decidable in Separation Logic Beyond Progress, Connectivity and Establishment?	157
<i>Tanguy Bozec, Nicolas Peltier, Quentin Petitjean, and Mihaela Sighireanu</i>	
Sequents vs Hypersequents for Åqvist Systems	176
<i>Agata Ciabattoni and Matteo Tesi</i>	
Uniform Substitution for Differential Refinement Logic	196
<i>Enguerrand Prebet and André Platzer</i>	

Sequent Systems on Undirected Graphs	216
<i>Matteo Acclavio</i>	
A Proof Theory of (ω -)Context-Free Languages, via Non-wellfounded Proofs	237
<i>Anupam Das and Abhishek De</i>	
A Cyclic Proof System for Guarded Kleene Algebra with Tests	257
<i>Jan Rooduijn, Dexter Kozen, and Alexandra Silva</i>	
Unification, Rewriting and Computational Models	
Unification in the Description Logic $\mathcal{ELH}\mathcal{R}^+$ Without the Top Concept Modulo Cycle-Restricted Ontologies	279
<i>Franz Baader and Oliver Fernández Gil</i>	
Confluence of Logically Constrained Rewrite Systems Revisited	298
<i>Jonas Schöpfung, Fabian Mitterwallner, and Aart Middeldorp</i>	
Equational Anti-unification over Absorption Theories	317
<i>Mauricio Ayala-Rincón, David M. Cerna, Andrés Felipe González Barragán, and Temur Kutsia</i>	
The Benefits of Diligence	338
<i>Victor Arrial, Giulio Guerrieri, and Delia Kesner</i>	
A Dependency Pair Framework for Relative Termination of Term Rewriting ...	360
<i>Jan-Christoph Kassing, Grigory Vartanyan, and Jürgen Giesl</i>	
Solving Quantitative Equations	381
<i>Georg Ehling and Temur Kutsia</i>	
Equivalence Checking of Quantum Circuits by Model Counting	401
<i>Jingyi Mei, Tim Coopmans, Marcello Bonsangue, and Alfons Laarman</i>	
Author Index	423

Invited Contributions



Automated Reasoning for Mathematics

Jeremy Avigad^(✉) 

Carnegie Mellon University, Pittsburgh, PA 15213, USA
avigad@cmu.edu

Abstract. Throughout the history of automated reasoning, mathematics has been viewed as a prototypical domain of application. It is therefore surprising that the technology has had almost no impact on mathematics to date and plays almost no role in the subject today. This article presents an optimistic view that the situation is about to change. It describes some recent developments in the Lean programming language and proof assistant that support this optimism, and it reflects on the role that automated reasoning can and should play in mathematics in the years to come.

1 The Origins and Foundations of Automated Reasoning

The fact that IJCAR is celebrating the 30th anniversary of the CADE ATP System Competition (CASC) is a reminder that, at least by the standards of computer science, automated reasoning has a long and venerable history. Some date the field to 1956, when Allen Newell, Herbert Simon, and Cliff Shaw introduced the *Logic Theorist*, a program that used heuristic methods to prove theorems in propositional logic. Two years earlier, however, Martin Davis had implemented Presburger's decision procedure for integer arithmetic on a computer at the Institute for Advanced Study. Davis admitted that the program did not perform well but he reported that it succeeded in proving that the sum of two even numbers is even. In 1960, Henry Gelernter, J. R. Hansen, and Donald Loveland published an article on the *Geometry Machine*, a program that could prove non-trivial theorems in elementary Euclidean plane geometry. The resolution rule for propositional logic was introduced by Davis and Hilary Putnam in 1960, and John Alan Robinson's introduction of a unification algorithm in 1965 established resolution theorem proving as a powerful method for first-order logic.¹

The theoretical foundations of automated reasoning predate even the introduction of the first electronic computers. In contemporary terms, Kurt Gödel's first incompleteness theorem says that there is no complete, consistent, computably axiomatized theory that contains (or interprets) a modicum of arithmetic. In his 1931 paper, Gödel explained that the theorem, as he stated it,

¹ All of the articles mentioned in this paragraph are found in a collection edited by Siegmund and Wrightson [50]. See also the survey by Mackenzie [36] and the references there.

is not in any way due to the special nature of the systems that have been set up, but holds for a wide class of formal systems; among these, in particular, are all systems that result from the two just mentioned through the addition of a finite number of axioms ... [21,22]

It is interesting to see Gödel struggling to say “computably axiomatized theory” for the simple reason that, at the time, there was no mathematical concept of computability available. He then presented a tentative definition of computability in lectures that he gave at the Institute for Advanced Study in Princeton in 1932 precisely so that he could state the incompleteness theorems in their proper generality. Alan Turing gave his own celebrated definition of computability a few years later and titled the paper “On computable numbers, with an application to the Entscheidungsproblem,” providing a negative answer to Hilbert’s question as to whether there is a decision procedure for first-order logic. Gödel had expressed uncertainty as to whether his definition exhausts the general notion of computability, but he took Turing’s analysis to settle the matter definitively:

In consequence of later advances, in particular of the fact that, due to A. M. Turing’s work, a precise and unquestionably adequate definition of the general concept of a formal system can now be given, the existence of undecidable arithmetical propositions and the non-demonstrability of the consistency of a system in the same system can now be proved rigorously for *every* consistent formal system containing a certain amount of finitary number theory. (quoted in [17])

The emphasis is by Gödel, who took the phrase “formal system” to mean a system with computably checkable axioms and rules.

Turing pointed out in his paper that the first incompleteness theorem is a consequence of the undecidability of theories of arithmetic, because if there were a complete, computably axiomatized theory of arithmetic one could decide the provability of a formula by simultaneously searching for a proof of the formula and its negation. Alonzo Church gave an independent proof of the undecidability of arithmetic in 1936, and Stephen Kleene, another key player in developing the modern theory of computability, was also keenly interested in applications to logic and the foundations of mathematics. So logicians were thinking about computable proof systems and proof search even before the arrival of the first digital electronic computers in the 1940s.

Fundamental decision procedures for logic and arithmetic also predate the electronic computer. As early as 1915, Leopold Löwenheim gave a decision procedure for monadic first-order logic. (The introduction to Börger et al. [11] provides an excellent overview of early work on the decidability of fragments of first-order reasoning.) Mojżesz Presburger’s paper on a decision procedure for arithmetic, based on Thoralf Skolem’s method of eliminating quantifiers, was published in 1929. (Presburger never earned a doctorate for that work; Andrzej Mostowski reported [15] that Alfred Tarski thought the result was too simple, a straightforward application of Thoralf Skolem’s method of elimination of quantifiers.) In 1930, Tarski obtained a decision procedure for real-closed fields, that is, the

first-order theory of the real numbers as an ordered field, though the result was not published until 1948. So logicians were also interested in decision procedures for aspects of mathematical reasoning even before there were computers to implement them.

2 Taking Stock

We have seen that the early history of automated reasoning was rooted in the foundations of mathematics and that many of its early pioneers were mathematicians. Even those who were not mathematicians took mathematical reasoning to be a primary target of the technology. Now, almost a century after Presburger's discovery of a decision procedure for arithmetic and almost three quarters of a century after the implementation of the Logic Theorist, it seems reasonable to ask where we stand. What has automated reasoning done for mathematics, and how is it used in mathematics today?

The answer is disappointingly negative. Automated reasoning has had almost no impact at all on mathematics and plays almost no role in the subject today. Few working mathematicians have ever touched an automated reasoning tool, let alone use automated reasoning in their daily work. The technology has contributed to very few mathematical discoveries, even minor ones.

This is surprising. One would think, as the pioneers of the subject clearly did, that mathematical reasoning is ideally suited to automation. To be sure, mathematics requires creativity, intuition, experience, and insight, but it also requires long chains of precise and sometimes tedious reasoning, and it's hard to get the details right. Computers can carry out small inferential steps much more quickly and accurately than we can, so we would expect them to be helpful for exploring and verifying mathematical results. Numeric and symbolic computation have revolutionized the sciences, even though science involves a lot more than computation. Why hasn't automated reasoning had a similar impact on mathematics?

This question is not meant as a criticism. Automated reasoning has made remarkable progress over the past 70 years, and the tools are now quite sophisticated. They have had a significant impact in several important areas, such as hardware and software verification, AI, planning, databases, knowledge representation, program synthesis, and natural language processing. Automated reasoning does not have to look to mathematics for justification. Moreover, the fact that there have been few applications to mathematics doesn't mean that there haven't been any, and the successes are worth celebrating. Finally, the fact that it has been difficult to automate mathematical reasoning is largely a reflection of the fact that mathematics isn't easy to mechanize, and those of us who love the subject wouldn't want it any other way. So my goal here is rather to review some of the successes of automated reasoning for mathematics, understand the challenges, and reflect on the role that automated reasoning can and should play in mathematics in the years to come.

In 2019, I gave a joint talk at FroCoS and TABLEAUX titled "Automated Reasoning for the Working Mathematician." In that talk, I surveyed the use of

automated reasoning with interactive proof assistants in the hopes of extracting lessons that I could convey to the automated reasoning community. This article draws on that talk as well as unpublished notes, data, and experiments that I prepared at the time.² See also my article, “The Mechanization of Mathematics” [1], and an article by Michael Beeson with the same title [6], for additional examples of applications of automated reasoning to mathematics.

3 A Personal History

As a mathematician who has been using automated reasoning tools for more than two decades, my interest in the subject is personal. I first experimented with Isabelle and Coq in the late 1990s, and when I started using Isabelle in earnest in 2002, the automation was surprisingly mature. There were a conditional term rewriter, `simp` [44], variations on a general tableau prover (`auto`, `force`, and `clarify` [45]), and a decision procedure for linear arithmetic (`arith`). Working with students at Carnegie Mellon, I completed a proof of the Hadamard–de la Vallée-Poussin prime number theorem in September of 2004 [2]. A couple of months later, Georges Gonthier announced the verification of the four-color theorem in Coq [24], and soon after that Thomas Hales announced the verification of the Jordan curve theorem in HOL Light [26]. These were early landmarks, providing evidence that substantial mathematical theorems could be formalized.

Many of the challenges in formalizing the prime number theorem stemmed from the fact that Isabelle’s libraries were young and incomplete. The automation, however, was remarkably helpful. For example, in the 4,000 lines contained in the last file in the proof, there are 390 invocations of `simp`, 397 invocations of `auto` and friends, and 246 invocations of `arith`. Even now, twenty years later, I have yet to have a better experience with automation.

I spent a sabbatical year in France with Gonthier and his team in 2009–2010, working on the formalization of the Feit–Thompson theorem, using the SSReflect proof language and Coq [25]. In designing and managing the project, Georges made the conscious decision to avoid automation entirely, other than the built-in foundational reduction of Coq expressions, which is fundamental to the methodology of SSReflect. He was skeptical that black-box automation would scale and had more faith in the power of good language design to make formalization manageable.

When I returned from France, I was ready to leave interactive theorem proving behind and turn to automated reasoning. But a talented undergraduate at Carnegie Mellon, Luke Serafin, managed to convince me to work on a project to verify the central limit theorem in Isabelle [3], and I was seduced by the excitement around homotopy type theory at the time to work on another verification project in Coq [4].

What really pulled me back to the world of proof assistants, however, was Leonardo de Moura’s decision, in 2013, to launch the Lean project. Leo convinced

² <https://github.com/avigad/arwm>.

me that even if one is primarily interested in automation for mathematics, one should build it on top of a secure, expressive foundation, not just to ensure that the automation is reliable, but to have a meaningful specification of what the results mean. For several years, Lean’s web pages described the aim of the project as follows:

to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs.

I don’t think Leo anticipated the amount of work he would have to put into implementing an elaborator for dependent type theory and supporting all the features that are needed to make that foundation usable. Work also went into the implementation of a tactic framework, in Lean 2, and the implementation of a metaprogramming language, in Lean 3, that users could use to write their own tactics [18, 39]. Lean 4 is a complete rewrite of the system, most of which is now implemented in Lean 4 itself [38]. Leo and Sebastian Ullrich have put a lot of effort into making Lean 4 an efficient programming (and metaprogramming) language, and treating syntax as first-class objects, making the syntactic framework as powerful, flexible, and extensible as the tactic framework.

We are now beginning to see automation for Lean 4 that is written in Lean 4, as well as Lean-based experiments on applications of machine learning to mathematical reasoning. Thus, a decade into the Lean project, we are now in an especially good position to realize the initial vision of making it a powerful means of combining automation with user interaction. In the sections that follow, I will discuss prospects for automated reasoning for mathematics in general, but I will also focus specifically on opportunities based on recent developments in Lean.

4 Domain-General Reasoning for Verification

To prepare for the talk at FroCoS and TABLEAUX, I sent a questionnaire to colleagues who had worked on formalization of mathematics to learn about their experiences with automation. One of the interesting findings was that most of the people who I considered to be the best at formalization—people who had formalized vast amounts of interesting mathematics efficiently and with very high quality—used very little automation at all. (Larry Paulson was a notable exception; he has spoken eloquently of the power of automation in enabling him to port large amounts of measure theory and analysis from HOL Light to Isabelle.) The best explanation I could come up with is that even if automation were much better than it is now, serious users would still have to fill in some inferences by hand, which would inevitably require them to learn the library inside out and become skillful at writing explicit proofs. So even when automation is available, power users generally come to know the library and proof language well enough that they don’t need to use automation to do what they want to do. If that analysis is correct, it highlights the challenge of scaling the use of formal methods to a broader mathematical audience. Even now, I get frustrated

when I have to struggle with an unfamiliar part of the library to carry out an inference that seems painfully obvious. The lack of automation limits the utility of formalization to all but the most determined and dedicated practitioners.

Let me clarify that when I talk about domain-general reasoning, I am setting aside equational rewriting and simplification. It’s not clear how to classify such methods. On the one hand, there is nothing more general than the equality relation: wherever there are expressions that denote objects, there are equations that govern them. On the other hand, equational rewriters handle only a small fragment of logical reasoning and the task they perform is focused and specific. If we take domain-general reasoning to encompass problems that, in full generality, are equivalent to the halting problem, it makes sense to exclude equational rewriters, which are designed to reduce expressions to canonical forms in a finite number of steps. In any case, they are incredibly useful. Tools like Isabelle’s `simp` can simplify a formula to `True` and hence prove more than just equations. They can also carry out conditional rewriting and use backchaining to dispel side conditions. Users can add facts to the rewrite database as they develop new theories. As far as I know, any proof assistant that takes automation seriously has some sort of rewrite engine. Lean’s version of `simp` was one of the first tactics that was implemented in that system.

To prepare for the talk at FroCoS and TABLEAUX, I also carried out some experiments with Isabelle’s *Sledgehammer* [46]. This is a tool that, given a proof goal, uses a relevance filter to select a couple of hundred potentially useful theorems from the library, sends the problem to external provers, and then tries to use the information they return to reconstruct a proof internally. I set myself the task of determining the extent to which I could formalize theorems by writing a proof sketch, calling only *Sledgehammer* and `auto`, and then refining the sketch as needed. I formalized three theorems in this way—the mutilated chessboard problem, the intermediate value theorem, and the existence of infinitely many primes congruent to three modulo four—and I took detailed notes as I went. The data, which is still available in the GitHub repository, is not very rigorous, but it helped me understand better some of the places where the automation fell short. In particular, two of the theorems required mild forms of second-order reasoning, such as identities governing the summation of functions over finite sets or reasoning about membership in sets defined by explicit predicates. Now, provers like Vampire, Zipperposition, and E can handle higher-order reasoning natively [7, 8, 53], which is likely to help.

One method of proof reconstruction involves harvesting nothing more than the list of theorems the external prover needed to establish the goal and calling internal, proof-producing automation to redo the search. Isabelle uses a tool called *Metis* for that [31]. Joshua Clune, Yicheng Qian, and Alexander Benkamp have written a proof-producing superposition theorem prover for Lean called *Duper* to serve a similar purpose, as well as to serve as generic internal automation. They invested considerable effort in adapting conventional resolution methods to dependent type theory. For example, in dependent type theory,

a data type might not have any elements, and Skolemization and other components of the search calculus have to be adapted to avoid introducing unsoundness. Duper can instantiate generic type variables on the fly but that introduces additional technical complications, as does adapting unification procedures to the dependently typed setting. Because it is modeled on Zipperposition, Duper can also handle higher-order inferences, and because it operates on Lean expressions directly (rather than via translation), it is possible to handle other rules tailored specifically to dependent type theory. Testing on standard benchmarks shows that Duper’s performance is roughly comparable to Metis’, offering hope that we will have a Sledgehammer for Lean before long. Lean is starting to catch up with Isabelle in other respects as well, with an automated reasoner called Aesop [33], inspired by Isabelle’s `auto`, as well as tools like Coq’s `eauto` and PVS’s `grind`.

When we consider the way that mathematicians use proof assistants, it becomes clear that support for reasoning about algebraic structures is essential. I have sometimes heard computer scientists say that there is no need to use dependent type theory because anything that can be done there can be done just as well in set theory or simple type theory. In principle, any reasonable foundation can interpret any other, possibly modulo a few axiomatic extensions, but generally speaking, the remark fails to appreciate the extent to which algebraic language and thought pervade contemporary mathematics. Any undergraduate student of mathematics can talk about the ring of $n \times n$ matrices of polynomials over $\mathbb{Z}/p\mathbb{Z}$ for a prime number p . Moreover, that student knows that multiplication is commutative on the polynomials and their coefficients but not on the matrices, and that nonzero elements of $\mathbb{Z}/p\mathbb{Z}$ have multiplicative inverses while the polynomials and matrices generally don’t. In other words, mathematicians easily name complex structures and use generic notation, and they know what properties elements of those structures have. The structures themselves are mathematical objects on par with numbers, functions, and circles, yet at the same time they serve as data types, constraining what can meaningfully be said about their elements. I find it remarkable that Lean and Mathlib are so good at making a vast network of structures available to users without collapsing under the technical requirements. Doing so requires a carefully designed system of type class inference and efficient means of elaborating and unifying the complex expressions that describe the structures and their elements. Dependent type theory may not be the only possible solution, but it is the only one implemented so far that can do anything close to what mathematicians need.

Reasoning about a rich algebraic hierarchy is a challenge for automated reasoning for at least two reasons: first, because instantiating generic theorems requires determining whether the types in question are instances of the relevant algebraic structures, and, second, because carrying out unification with expressions that have algebraic components requires determining the identity of objects that have been described in different ways. Both of these tasks are too expensive to be carried out in the midst of an automated search, but, fortunately,

we generally don't expect them to be: mathematicians are usually careful to establish the relevant algebraic context explicitly. Duper manages algebraic reasoning using a remarkable preprocessing tool by Qian called *Lean-Auto*, which heuristically instantiates generic theorems, infers the relevant algebraic structures, and chooses canonical representations of expressions, all before the search begins.

Sledgehammers generally work by translating the source language to a simpler one and using tools optimized for equational reasoning, propositional reasoning, and quantifier instantiation. Another approach to automating dependent type theory is to search systematically in dependent type theory itself. There are tools for Coq [16] and Agda [34, 51] that take this approach, without making any attempt at completeness. At Carnegie Mellon, Chase Norman has implemented a procedure for a minimal but fully expressive dependent type theory that provides a complete solution to both the unification and type inhabitation problems, generalizing Huet's unification procedure for higher-order logic [30]. The framework is flexible enough to instantiate various heuristics to carry out the search, and the implementation performs well on examples. It will be interesting to see whether such an approach will provide automation that complements the strengths of a sledgehammer.

5 Domain-Specific Reasoning for Verification

Talking about domain-general automation reminds me of a quip that I once heard attributed to Sidney Morgenbesser that philosophers are people who know something about everything but nothing about anything. In an ornery mood, I might complain that first-order theorem provers are good at reasoning about everything but not so good at reasoning about anything in particular. At the opposite end of the spectrum are domain-specific automated reasoning tools that carry out more deterministic and focused tasks, such as reasoning about arithmetic, establishing algebraic identities, reasoning about linear and nonlinear inequalities, and so on.

Tools like these are extremely useful. Lean has benefited from the availability of a metaprogramming language, introduced in Lean 3 [18] and made vastly more powerful in Lean 4 [38], that allows users to write tactics within Lean. The ability to attract mathematical users from 2017 on was bolstered by the fact that users like Mario Carneiro were able to quickly provide them with the tactics they needed. As mathematicians gained expertise with the system, they could design tactics that would help them in their daily work. For example, Heather Macbeth, with the help of Carneiro, wrote tactics `gcongr` and `positivity` to help with common calculations, and then could easily shorten a proof like this:

```
calc ||wp - wq|| * ||wp - wq||
  _ = 2 * (||a|| * ||a|| + ||b|| * ||b||) - 4 * ||u - half · (wq + wp)|| *
    ||u - half · (wq + wp)|| := by rw [← this]; simp
  _ ≤ 2 * (||a|| * ||a|| + ||b|| * ||b||) - 4 * δ * δ :=
    (sub_le_sub_left eq1 _)
  _ ≤ 2 * ((δ + div) * (δ + div) + (δ + div) * (δ + div)) -
```

```

      4 * δ * δ :=
      (sub_le_sub_right (mul_le_mul_of_nonneg_left
        (add_le_add eq1 eq2') (by norm_num)) _)
    _ = 8 * δ * div + 4 * div * div := by ring
exact
  add_nonneg (mul_nonneg (mul_nonneg (by norm_num) zero_le_δ)
    (le_of_lt Nat.one_div_pos_of_nat))
    (mul_nonneg (mul_nonneg (by norm_num)
      Nat.one_div_pos_of_nat.le) Nat.one_div_pos_of_nat.le)

```

to this:

```

calc ||wp - wq|| * ||wp - wq||
  _ = 2 * (||a|| * ||a|| + ||b|| * ||b||) - 4 * ||u - half · (wq + wp)|| *
    ||u - half · (wq + wp)|| := by simp [← this]
  _ ≤ 2 * (||a|| * ||a|| + ||b|| * ||b||) - 4 * δ * δ := by gcongr
  _ ≤ 2 * ((δ + div) * (δ + div) + (δ + div) * (δ + div)) -
    4 * δ * δ := by gcongr
  _ = 8 * δ * div + 4 * div * div := by ring
positivity

```

Tomáš Skřivan recently contributed a tactic, `fun_prop`, that effectively establishes properties like continuity, differentiability, and measurability of functions.

I am grateful to Adam Topaz for writing a small Lean metaprogram to extract tactic usage statistics from a recent version of Mathlib. The data is messy because tactic variants are listed separately when they are called under separate wrappers, including variants that were written to support the port of the library from Lean 3 but are otherwise superseded by newer versions. It is also misleading in that some tactics have been around much longer than others, so the numbers do not reflect the current utility. Nor does the data say anything about the role of domain-specific automation outside of Mathlib. Finally, some tactics are used transiently and are then eliminated from the final proof document, such as those that help find theorems, display information, or write proofs. These do not appear in the list.

Nonetheless, the data is informative. Tactics used to apply theorems are most common (`apply`, `exact`, `refine`, etc., with about 60K instances in all). After that, the vast majority of tactic calls, by far, are used for equational reasoning, with more than 52K invocations of Lean’s `rw` tactic, which does manual term rewriting, and more than 60K invocations of Lean’s simplifier (`simp`, `simp_a`, `dsimp`, and `simp_rw`). About 25K invocations are used to decompose data and existential assertions (`obtain`, `rintro`, `rcases`, `cases`, etc.), and there are about 5K calls to tactics that carry out proof by induction.

More specialized automation still manages to carry its weight. The `linarith` tactic is called more than 1,100 times; `split_ifs`, which splits a goal to simplify conditional expressions, and `ring`, which carries out ring calculations, are each called more than 1,000 times; `filter_upwards`, a simple tactic that helps reason about filters in measure theory and analysis is called almost 900 times; `norm_num`, which does numeric calculations, is called more than 800 times; a specialization of Aesop for use with category theory, `aesop_cat`, is called almost 800 times; `positivity`, a relative newcomer, is called more than 600 times; and `norm_cast`,

a tactic to help mediate casts between numeric domains, and `gcongr` are each called more than 500 times.

Lean’s metaprogramming language also provides flexible ways to support better interactions with automation, both domain-general and domain-specific. Lean’s *Widgets* framework [42] allows users to install custom Javascript-driven displays of objects and information in Lean’s “infoview” window in VS Code, and allows user interactions with these graphical displays to communicate information and actions back to Lean and the editor. When the user puts the cursor at the beginning of a tactic invocation in a proof, the infoview window highlights the part of the state that is about to change, and when the cursor is on or at the end of the invocation, it highlights what has just changed. Users can hover over constants in the infoview window to see documentation, they can click on expressions to see their types, and they can control-click on identifiers to jump to their definitions. Users can trace class inference to diagnose failures, and expand or collapse nodes of the search tree in the infoview window. In Lean, automation can return structured error messages that can be explored. Ideally, whenever automation fails, users should have the means to diagnose the problem and come up with suitable interventions to fix it. Developers of automation should keep user interfaces in mind both as targets for automated reasoning and as means for using automation more effectively.

Finally, it is worth mentioning that tools that help users find theorems and explore the library are also essential. Lean provides internal tactics like `apply?`, `exact?`, and `rw?` that suggest atomic proof steps. There is also a good symbolic search engine, *Loogle*, and there is another search tool, *Moogle*, that uses a large language model to answer natural-language queries.

6 Automation for the Discovery of New Theorems

To this point, we have focused on the use of automated reasoning to verify mathematical results that were discovered by conventional means. Mathematicians, however, tend to be much more excited about methods that help with the discovery of new mathematics. The automated reasoning community is justifiably proud of William McCune’s use of his theorem prover, EQP, to settle the Robbins conjecture in 1996 [37]. The result, which shows that a certain set of equations can be taken as an alternative axiomatization of Boolean algebras, made the pages of the *New York Times*. Since then, there has also been a small industry in using automated theorem provers to prove theorems about other algebraic structures, like loops and quasigroups. One can think of a loop as like a group except without the associativity axiom, and one can think of a quasigroup as a loop without an identity. First-order theorem provers have been used to establish consequences of these and related axioms, an industry nicely surveyed by Phillips and Stanovský [47].

I have heard mathematicians express annoyance, however, at the suggestion that these results have anything to do with real mathematics. Few mathematicians have heard of the Robbins conjecture or have any interest in quasigroups.

More to the point, mathematicians chafe at the implication that modern algebra is about deriving first-order consequences of axioms. Algebraists are interested in classification theorems, which characterize structures in terms of key invariants, structure theorems, which provide means of understanding structures in terms of subobjects and morphisms to other structures, and representation theorems. They are interested in introducing new structures and new spaces of structures, with applications that explain and simplify past results and provide powerful tools for future research. All these involve reasoning about structures within the context of a rich mathematical theory, rather than reasoning deductively from the axioms. As a result, to most mathematicians, the applications of automated reasoning to algebra so far are little more than recreational curiosities.

Applications of SAT solvers to mathematics fare better. For example, in 1912, Issai Schur proved that given any finite coloring of the positive integers, there is a monochromatic solution to the equation $x + y = z$. Today, this is recognized as a seminal result in both Ramsey theory and additive number theory. The theorem raises the question as to whether one can compute the largest initial segment of the positive integers $\{1, 2, \dots, S(k)\}$ such that there is a k -coloring with no such monochromatic solution. It's not hard to establish $S(1) = 1$, $S(2) = 4$, and $S(3) = 13$. In 1965, Golomb and Baumert computed $S(4) = 44$ in a paper that contains other interesting examples of backtracking search [23]. The value $S(5) = 160$ was computed by Heule in 2017 using a SAT solver [28], a result which has drawn praise from the mathematical community. The value of $S(6)$ is still unknown.

Most mathematicians aren't interested in calculating Schur numbers, but the problem is considered at least interesting by association, given that they recognize Schur's theorem as an important theorem. The case is similar with respect to a theorem that Paul Erdős dubbed the "happy ending problem" because it led to the marriage of George Szekeres and Esther Klein. The general version of the theorem says that for every positive integer n , any sufficiently large finite set of points in general position in the plane contains a convex n -gon. Let $f(n)$ denote the smallest number of points in general position that provides such a guarantee; the value of $f(n)$ is known only for $n \leq 6$. A related problem asks for the smallest number of points in general position that guarantees the existence of an *empty* convex n -gon, that is, one without any points in its interior. There are infinite sets of points without a convex 7-gon, but Nicolás [43] and Gerken [20] proved independently that every sufficiently large set of points in general position contains an empty convex hexagon. Using a SAT solver, Heule and Scheucher [29] recently showed that 30 is the smallest number of points that provides that guarantee.

When SAT solvers are used to solve mathematical problems, it is important to have guarantees that the results are correct. Students at Carnegie Mellon are working on a SAT library for Lean that addresses that concern. Joshua Clune has written an LRAT checker that is currently in use at Amazon Web Services; Wojciech Nawrocki has verified a checker for *knowledge compilation*, a technology that, in particular, can provide precise counts of the number of satisfying

assignments to a propositional formula [12]; and Cayden Codel has written a verified checker for SR, a strong proof format for SAT solvers that incorporates symmetries in “without loss of generality” reductions [13]. Perhaps the more serious concern is to verify that a problem that is sent to a SAT solver is a correct representation of the intended problem. This is pressing because the reduction of an ordinary mathematical statement to a SAT problem often relies on complex encodings as well as symmetry breaking and other reductions, and the generation of the propositional formula is further subject to subtle programming errors. Codel, Nawrocki, and James Gallicchio have been working on aspects of the library that address that problem as well [14]. Bernardo Subercaseaux, Nawrocki, Gallicchio, Codel, Carneiro, and Heule have verified the correctness of the encoding used to compute the empty hexagon number [52].

Mathematicians have yet to explore the use of automated reasoning tools to *find* objects of mathematical interests. SAT solvers, SMT solvers, constraint solvers, and model finders are all designed to find objects and structures satisfying given constraints. A popular Isabelle tool called *Nitpick* [10] uses a model finder to look for counterexamples to purported theorems, in order to prevent them from investing time and energy in trying to prove a statement that is false.

Bespoke decision procedures can also aid the process of discovery. An automated reasoning tool called *Walnut* [40, 49] implements a decision procedure for an extension of Presburger arithmetic that can express properties of *automatic sequences*, which are, roughly, sequences generated by finite state automata. Consider the question as to whether there is an infinite binary sequence with no subsequence of the form xxx^R , where x is a finite sequence and x^R is its reversal. It is not hard to see that the sequence 01010101... has that property, but is it possible to find one that is aperiodic? A paper by Mousavi, Schaeffer, and Shallit [41] explains how Walnut helped them construct such a sequence.

I believe that mathematicians’ general habit of dismissing “finite problems” as not properly mathematical will change over time. The entire edifice of infinitary mathematics bears on our everyday experience only through measurement and observation, and discrete problems from computer science have already begun to influence mathematical research. It also seems likely that mathematicians will find creative ways to solve infinitary problems by devising representations and reductions that are amenable to automation. The main challenge is that automated reasoning is unfamiliar to them. The history of Lean suggests that mathematicians will go to extraordinary lengths to learn a new technology once they decide that it is interesting and aligns with their goals. To facilitate adoption, it helps to have documentation and expository materials that are written with them in mind. The incentive structures in mathematics and computer science are not good at encouraging that kind of cross-disciplinary outreach, but once the door is open, it is only a matter of time before a new technology becomes part of the mathematical mainstream.

7 Machine Learning and Symbolic AI

It’s impossible to write about the prospects of automated reasoning for mathematics today without saying something about machine learning. Machine learning and symbolic AI have complementary strengths: ML is good at synthesizing vast amounts of data but isn’t good at getting details right, whereas symbolic methods are good at getting the details right but are overwhelmed by combinatorial explosion in a search space. A central challenge for AI is to design systems that get the best of both worlds by combining the two approaches, and mathematics, where the problems are especially clear and well-defined, is an ideal place to make progress.

It is therefore not surprising that there is growing interest in applications of machine learning to mathematics. Researchers have long explored the use of machine learning techniques to guide symbolic search and to select premises for a sledgehammer, and with the advent of deep learning, there has been a surge of interest in using neural networks to prove theorems in conjunction with an interactive proof assistant. A recent “brief survey” of machine learning in automated reasoning by Blaauwbroek et al. [9] has 168 references, and surveys of deep learning for mathematical reasoning by Lu et al. [35] and by Li et al. [32] have well over 200 references each.

Lean is becoming recognized as an ideal platform for such work. There have been at least two projects on using machine learning for premise selection for Lean [19, 48], there are tactics and code pilots based on Lean [27, 54, 55], and there are tools that support data extraction and interaction with Lean for machine learning experiments, including one developed by Kaiyu Yang and coauthors [55] and two developed by Kim Morrison.³ The MiniF2F problem benchmark [56] includes versions for Lean 3, and the ProofNet benchmarks [5] have recently been ported to Lean 4 by Abhijit Chakraborty and Rahul Vishwakarma.⁴ The features that make Lean a good platform for automated reasoning also make it a good platform for machine learning and support a synthesis of the two. The size and sophistication of Lean’s mathematical library, Mathlib, and the involvement of the mathematical community provide powerful opportunities for progress.

8 Conclusions

Although I began with a disappointing assessment of the current state of automated reasoning for mathematics, I hope I have conveyed good reasons for optimism. Mathematicians are beginning to warm to the use of formal methods, opening up new avenues for progress. As the technology improves, the number of mathematicians making use of the automated reasoning tools will increase, providing greater incentives for computer scientists to focus on mathematical applications. This, in turn, will increase the number of mathematicians who use

³ <https://github.com/semorrison/lean-training-data>
<https://github.com/leanprover-community/repl>.

⁴ <https://github.com/rahul3613/ProofNet-lean4>.

the technology and can therefore provide feedback and even contribute to its development. In short, I expect that we are on the cusp of a virtuous cycle whereby technological improvements lead to more users, which, in turn, lead to further improvements.

In this article, I have tried to identify some of the key challenges to developing better automation for mathematics, and I have suggested specific approaches that I find promising. The biggest challenges, however, may be sociological rather than technical. Making automation useful for mathematics will require mathematicians and computer scientists working together, and neither discipline will get far on its own. Mathematicians and computer scientists have very different attitudes and outlooks. Computer scientists focus on disseminating information quickly in conference publications, and their success is measured by the number of citations they receive. With the weight of centuries behind them, mathematicians can't be rushed, are mistrustful of academic fads, and tend to look to the leading experts in their fields to determine what is important. Whereas computer scientists value measurable impact in the short term, mathematicians answer to less clear-cut assessments of the quality and depth of their work. It's not that one discipline's standards are better than the other; each has its advantages and problems. It's just that the disparity of outlooks often makes communication difficult.

I'd like to suggest to computer scientists reading this article that it might be nice to adopt a mathematical attitude every once in a while. Imagine thinking about something because you find it interesting, without caring what others think. Imagine exploring ideas to see where they take you, without worrying about whether that will result in a conference publication by the next round of deadlines. Imagine working on a problem just for the joy of taking up the challenge. If all that sounds good to you, you'll find mathematicians to be excellent companions. I am not suggesting that you should turn your back on computer science; of course, you will still have bills to pay. But I am confident that one day, when you look back over your career, any contributions you have made to mathematics will be among the things that you are most proud of, and among those that are closest to your heart. So I invite you to come to the Lean Zulip channel and start talking to mathematicians about the things that automation can do for them. I promise, you won't regret it.

References

1. Avigad, J.: The mechanization of mathematics. *Notices Amer. Math. Soc.* **65**(6), 681–690 (2018). <https://doi.org/10.1090/noti1688>
2. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.* **9**(1), 2 (2007). <https://doi.org/10.1145/1297658.1297660>
3. Avigad, J., Hölzl, J., Serafin, L.: A formally verified proof of the central limit theorem. *J. Autom. Reason.* **59**(4), 389–423 (2017). <https://doi.org/10.1007/S10817-017-9404-X>

4. Avigad, J., Kapulkin, K., Lumsdaine, P.L.: Homotopy limits in type theory. *Math. Struct. Comput. Sci.* **25**(5), 1040–1070 (2015). <https://doi.org/10.1017/S0960129514000498>
5. Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E.W., Radev, D., Avigad, J.: Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *CoRR arXiv: 2302.12433* (2023). <https://doi.org/10.48550/ARXIV.2302.12433>
6. Beeson, M.J.: The mechanization of mathematics. In: Alan Turing: life and legacy of a great thinker, pp. 77–134. Springer, Berlin (2004)
7. Bentkamp, A., Blanchette, J., Nummelin, V., Turret, S., Vukmirovic, P., Waldmann, U.: Mechanical mathematicians. *Commun. ACM* **66**(4), 80–90 (2023). <https://doi.org/10.1145/3557998>
8. Bentkamp, A., Blanchette, J., Turret, S., Vukmirović, P.: Superposition for higher-order logic. *J. Autom. Reasoning* **67**(1) (2023). <https://doi.org/10.1007/s10817-022-09649-9>
9. Blaauwbroek, L., et al.: Learning guided automated reasoning: a brief survey. *CoRR arXiv: 2403.04017* (2024)
10. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) *ITP 2010*. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_11
11. Börger, E., Grädel, E., Gurevich, Y.: The classical decision problem. Universitext. Springer-Verlag, Berlin (2001), reprint of the 1997 original
12. Bryant, R.E., Nawrocki, W., Avigad, J., Heule, M.J.H.: Certified knowledge compilation with application to verified model counting. In: Mahajan, M., Slivovsky, F. (eds.) *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023*, 4-8 July 2023, Alghero, Italy. *LIPICs*, vol. 271, pp. 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICs.SAT.2023.6>
13. Buss, S., Thapen, N.: DRAT and propagation redundancy proofs without new variables. *Log. Methods Comput. Sci.* **17**(2) (2021). [https://doi.org/10.23638/LMCS-17\(2:12\)2021](https://doi.org/10.23638/LMCS-17(2:12)2021)
14. Codel, C.R., Avigad, J., Heule, M.J.H.: Verified encodings for SAT solvers. In: Nadel, A., Rozier, K.Y. (eds.) *Formal Methods in Computer-Aided Design, FMCAD 2023*, Ames, IA, USA, 24-27 October 2023, pp. 141–151. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_22
15. Crossley, J.N.: Reminiscences of logicians. In: Crossley, J.N. (ed.) *Algebra and Logic*, pp. 1–62. Springer, Berlin (1975)
16. Czajka, Ł: Practical proof search for coq by type inhabitation. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *IJCAR 2020*. LNCS (LNAI), vol. 12167, pp. 28–57. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_3
17. Davis, M.: *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvability Problems, and Computable Functions*. Dover Publications, revised edition edn. (2004)
18. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.* **1**(ICFP), 34:1–34:29 (2017). <https://doi.org/10.1145/3110278>
19. Geesing, A.: *Premise Selection for Lean 4*. Master’s thesis, Vrije Universiteit Amsterdam (2023)
20. Gerken, T.: Empty convex hexagons in planar point sets. *Discrete Comput. Geom.* **39**(1–3), 239–272 (2008). <https://doi.org/10.1007/s00454-007-9018-x>

21. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. Math. Phys.* **38**(1), 173–198 (1931). <https://doi.org/10.1007/BF01700692>, English translation in [22]
22. Gödel, K.: *Collected works*. Vol. I. Oxford University Press, New York (1986), edited by Feferman, S., Dawson Jr., J.W., Kleene, S.C., Moore, G.H., Solovay, R.M., van Heijenoort, J
23. Golomb, S.W., Baumert, L.D.: Backtrack programming. *J. Assoc. Comput. Mach.* **12**, 516–524 (1965). <https://doi.org/10.1145/321296.321300>
24. Gonthier, G.: Formal proof—the four-color theorem. *Notices Amer. Math. Soc.* **55**(11), 1382–1393 (2008)
25. Gonthier, G., et al.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) *ITP 2013*. LNCS, vol. 7998, pp. 163–179. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_14
26. Hales, T.C.: The Jordan curve theorem, formally and informally. *Amer. Math. Monthly* **114**(10), 882–894 (2007). <https://doi.org/10.1080/00029890.2007.11920481>
27. Han, J.M., Rute, J., Wu, Y., Ayers, E.W., Polu, S.: Proof artifact co-training for theorem proving with language models. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, 25-29 April 2022*. OpenReview.net (2022), <https://openreview.net/forum?id=rpxJc9j04U>
28. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI 2018), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI 2018)*, New Orleans, Louisiana, USA, 2-7 February 2018, pp. 6598–6606. AAAI Press (2018). <https://doi.org/10.1609/AAAI.V32I1.12209>
29. Heule, M.J.H., Scheucher, M.: Happy ending: An empty hexagon in every set of 30 points (2024), to appear in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2024*
30. Huet, G.P.: A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.* **1**(1), 27–57 (1975). [https://doi.org/10.1016/0304-3975\(75\)90011-0](https://doi.org/10.1016/0304-3975(75)90011-0)
31. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pp. 56–68 (2003). <http://www.gilith.com/papers>
32. Li, Z., et al.: A survey on deep learning for theorem proving. *CoRR* (2024). <https://doi.org/10.48550/arXiv.2404.09939>
33. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: Krebbers, R., Traytel, D., Pientka, B., Zdancewic, S. (eds.) *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, 16-17 January 2023*, pp. 253–266. ACM (2023). <https://doi.org/10.1145/3573105.3575671>
34. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) *TYPES 2004*. LNCS, vol. 3839, pp. 154–169. Springer, Heidelberg (2006). https://doi.org/10.1007/11617990_10
35. Lu, P., Qiu, L., Yu, W., Welleck, S., Chang, K.: A survey of deep learning for mathematical reasoning. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (eds.) *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, 9-14 July 2023*, pp. 14605–14631. Association for Computational Linguistics (2023). <https://doi.org/10.18653/V1/2023.ACL-LONG.817>

36. Mackenzie, D.: The automation of proof: a historical and sociological exploration. *IEEE Ann. Hist. Comput.* **17**(3), 7–29 (1995). <https://doi.org/10.1109/85.397057>
37. McCune, W.: Solution of the Robbins problem. *J. Autom. Reason.* **19**(3), 263–276 (1997). <https://doi.org/10.1023/A:1005843212881>
38. Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) *CADE 2021*. LNCS (LNAI), vol. 12699, pp. 625–635. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_37
39. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) *CADE 2015*. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
40. Mousavi, H.: Automatic theorem proving in Walnut. *CoRR* (2016). <https://doi.org/10.48550/arXiv.1603.06017>
41. Mousavi, H., Schaeffer, L., Shallit, J.O.: Decision algorithms for Fibonacci-automatic words, I: basic results. *RAIRO Theor. Informatics Appl.* **50**(1), 39–66 (2016). <https://doi.org/10.1051/ITA/2016010>
42. Nawrocki, W., Ayers, E.W., Ebner, G.: An extensible user interface for Lean 4. In: Naumowicz, A., Thiemann, R. (eds.) *14th International Conference on Interactive Theorem Proving, ITP 2023*, 31 July to 4 August 2023, Białystok, Poland. *LIPICs*, vol. 268, pp. 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICs.ITP.2023.24>
43. Nicolás, C.M.: The empty hexagon theorem. *Discrete Comput. Geom.* **38**(2), 389–397 (2007). <https://doi.org/10.1007/s00454-007-1343-6>
44. Nipkow, T.: Term rewriting and beyond - theorem proving in Isabelle. *Formal Aspects Comput.* **1**(4), 320–338 (1989). <https://doi.org/10.1007/BF01887212>
45. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. *J. Univers. Comput. Sci.* **5**(3), 73–87 (1999). <https://doi.org/10.3217/JUCS-005-03-0073>
46. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) *The 8th International Workshop on the Implementation of Logics, IWIL 2010*, Yogyakarta, Indonesia, 9 October 2011. *EPiC Series in Computing*, vol. 2, pp. 1–11. EasyChair (2010). <https://doi.org/10.29007/36DT>
47. Phillips, J.D., Stanovský, D.: Automated theorem proving in quasigroup and loop theory. *AI Commun.* **23**(2–3), 267–283 (2010). <https://doi.org/10.3233/AIC-2010-0460>
48. Piotrowski, B., Mir, R.F., Ayers, E.W.: Machine-learned premise selection for Lean. In: Ramanayake, R., Urban, J. (eds.) *Automated Reasoning with Analytic Tableaux and Related Methods - 32nd International Conference, TABLEAUX 2023*. LNCS, vol. 14278, pp. 175–186. Springer (2023). https://doi.org/10.1007/978-3-031-43513-3_10
49. Shallit, J.: The logical approach to automatic sequences—exploring combinatorics on words with Walnut, London Mathematical Society Lecture Note Series, vol. 482. Cambridge University Press, Cambridge (2023). <https://doi.org/10.1017/9781108775267>
50. Siekmann, J., Wrightson, G. (eds.): *Automation of Reasoning 1*. Springer, Berlin (1983)
51. Skystedt, L.: A New Synthesis Tool for Agda. Master’s thesis, University of Gothenburg and Chalmers University of Technology (2022)
52. Subercaseaux, B., Nawrocki, W., Gallicchio, J., Codel, C., Carneiro, M., Heule, M.J.H.: Formal verification of the empty hexagon number (2024). <https://arxiv.org/abs/2403.17370>

53. Vukmirovic, P., Blanchette, J., Schulz, S.: Extending a high-performance prover to higher-order logic. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Part II. LNCS, vol. 13994, pp. 111–129. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_10
54. Welleck, S., Saha, R.: LLMSTEP: LLM proofstep suggestions in Lean. CoRR [arXiv: abs/2310.18457](https://arxiv.org/abs/2310.18457) (2023). <https://doi.org/10.48550/arxiv.2310.18457>
55. Yang, K., et al.: Leandojo: theorem proving with retrieval-augmented language models. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, 10 - 16 December (2023). http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html
56. Zheng, K., Han, J.M., Polu, S.: MiniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, 25-29 April 2022. OpenReview.net (2022). <https://openreview.net/forum?id=9ZPegFuFTFv>

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

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





Induction in Saturation

Laura Kovács¹  , Petra Hozzová¹ , Márton Hajdu¹ ,
and Andrei Voronkov^{2,3}

¹ TU Wien, Vienna, Austria

`laura.kovacs@tuwien.ac.at`

² University of Manchester, Manchester, UK

³ EasyChair, Manchester, UK

Abstract. Proof by induction is commonplace in modern mathematics and computational logic. This paper overviews and discusses our recent results in turning saturation-based first-order theorem proving into a powerful framework for automating inductive reasoning. We formalize applications of induction as new inference rules of the saturation process, add instances of appropriate induction schemata to the search space, and use these rules and instances immediately upon their addition for the purpose of guiding induction. Our results show, for example, that many problems from formal verification and mathematical theories can now be solved completely automatically using a first-order theorem prover.

1 Introduction

Proof by induction is commonplace in modern mathematics and computational logic. Many number-theoretic arguments rely upon mathematical induction over the natural numbers, while showing correctness of software systems typically requires structural induction over inductively-defined data types, to only name two examples. The wider automation of mathematics, logic, verification and other efforts therefore demands automating induction.

Induction can be automated by reducing goals to subgoals [1, 11], so that proving a goal $\forall x.F(x)$ can be proved by induction on x . However, splitting goals into subgoals and organizing proof search accordingly requires expert guidance. As an alternative, inductive reasoning has recently appeared in SMT solvers [13] and first-order theorem provers [3, 12, 14], complementing strong support for reasoning with theories and quantifiers. These approaches do not reduce goals to subgoals but instead implement tailored instantiations of induction schemas [3, 12, 14], adjust the underlying calculus with inductive generalizations [4] and function rewriting [6], extend theory reasoning for proving inductive formulas [8], and integrate induction with rewriting for generating auxiliary inductive properties during proof search [5].

This paper describes our recent efforts in these directions, entering new grounds in the automation of inductive reasoning. The distinctive feature of our work comes with mechanizing mathematical induction in saturation-based first-order theorem proving, turning thus saturation-based proof search into a

powerful framework to reason about software technologies, in particular about inductive properties of functional and imperative programs.

2 Induction in Saturation - In a Nutshell

Our work combines very efficient superposition-based equational reasoning with inductive reasoning, by extending superposition with new inference rules capturing inductive steps within saturation. We refer to these inference rules as *induction rules* and consider them in addition to superposition inferences during proof-search. Following the approach of [12], we capture the application of induction via the following general induction rule:

$$\frac{\bar{L}[t] \vee C}{F \rightarrow \forall x.L[x]} \text{ (Ind)},$$

where $L[t]$ is a ground literal, C is a clause, and $F \rightarrow \forall x.L[x]$ is a valid induction schema. Further, $\bar{L}[t]$ denotes the negation of $L[t]$.

In our work, we consider extensions and variants of the induction rule *Ind*, in order to add instances of appropriate induction schemata over inductive formulas to be proved. We call these instances *induction axioms* and automate induction in saturation via the following two, inter-connected steps:

- (i) devise new induction rules;
- (ii) optimize the saturation process with induction.

For step (i), we pick up a formula G in the search space and use induction rules to add new induction axioms Ax to the search space, aiming at proving $\neg G$, or sometimes a formula more general than $\neg G$. While our inference rules implement inductive reasoning upon G using Ax , adding only these inference rules to superposition-based proof search would be insufficient for efficient theorem proving. Modern saturation-based theorem provers are very powerful not just because of the logical calculi they are based on, such as superposition. What makes them powerful and efficient are redundancy criteria and pruning of the search space; strategies for directing proof search, mainly by clause and inference selection; and theory-specific reasoning, for built-in support for data types [8]. Therefore, in addition to devising new induction rules in (i), in (ii) we bring redundancy elimination, proof search options and theory axioms/rules to saturation with induction.

As a result of the combined efforts of (i)–(ii), induction in saturation maintains efficiency of standard saturation and is not limited to induction over specific (well-founded) theories. Such a genericity is particularly important for applying our results in the formal analysis of system requirements. For example, proving that every element in the computer memory is initialized or that no execution of a user request interferes with another user request, typically requires inductive reasoning with integers and arrays.

In the rest of the paper, we illustrate the automation of induction in saturation within the following three use-cases:

<pre> fun sum($n : \mathbb{Z}$) = if $n = 0$ then 0 else $n + \text{sum}(n - 1)$; <u>assert</u> $\forall n \in \mathbb{Z}. (n \geq 0 \rightarrow$ $2 \cdot \text{sum}(n) = n \cdot (n + 1))$ </pre> <p>(a) Sum of the first n positive integers.</p>	<pre> fun sum_evsg($n : \mathbb{Z}$) = if $n = 0$ then 0 else $(2 \cdot n)^2 + \text{sum_evsg}(n - 1)$; <u>assert</u> $\forall n \in \mathbb{Z}. (n \geq 0 \rightarrow$ $3 \cdot \text{sum_evsg}(n) = 2 \cdot n \cdot (n + 1) \cdot (2 \cdot n + 1))$ </pre> <p>(b) Sum of squares of the first n positive even integers.</p>
---	--

Fig. 1. Inductive reasoning with integers.

- proving arithmetical properties in Sect. 3;
- enforcing safety assertions of array-manipulating programs in Sect. 4;
- reasoning about the functional correctness of programs over lists in Sect. 5.

3 Induction and Arithmetic

We first discuss our work in proving inductive properties over (sums of) integers. While integers with the standard $<$ -ordering are not well-founded, we show that we can apply, and automate, induction over any integer interval with a finite bound [7].

In the sequel, we assume a distinguished *integer sort*, denoted by \mathbb{Z} . When we use standard integer predicates $<$, \leq , $>$, \geq , functions $+$, $-$, \dots and constants $0, 1, 2, \dots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations. All other symbols are uninterpreted. We will write quantifiers like $\forall x \in \mathbb{Z}$ to denote that x has the integer sort.

Example of Induction over Integers. Consider the recursive function `sum` of Fig. 1(a), computing the sum of the integers from the integer interval $[0, n]$. We aim to prove the assertion of Fig. 1(a), denoted via **assert** and stating that the value computed by `sum` is the closed-form expression describing the sum of the first n positive integers.

In order to prove the assertion of Fig. 1(a) within saturation-based proof search, we proceed as follows. We convert the function definition of `sum` into first-order axioms and negate the assertion of Fig. 1(a), skolemizing n as σ . We obtain the following unit clauses, with each clause being implicitly universally quantified:

$$\text{sum}(0) = 0 \tag{1}$$

$$n = 0 \vee \text{sum}(n) = n + \text{sum}(n - 1) \tag{2}$$

$$\sigma \geq 0 \tag{3}$$

$$2 \cdot \text{sum}(\sigma) \neq \sigma \cdot (\sigma + 1) \tag{4}$$

Clauses (1)–(2) result from the functional definition of `sum`, whereas clauses (3)–(4) yield the classified negation of the assertion of Fig. 1(a). We then continue

by applying inference rules on these clauses with the goal of refuting the negated assertion by deriving the empty clause, corresponding to a contradiction.

Induction Rule over Integers. When considering integers, we adjust the general induction rule **Ind** by considering induction over well-founded (integer) intervals. In particular, for proving property (4) of Fig. 1(a), we use the following extension of the **Ind** rule, where b is a ground term of integer sort:

$$\frac{\overline{L}[t] \vee C \quad t \geq b}{L[b] \wedge \forall y.(y \geq b \wedge L[y] \rightarrow L[y+1]) \rightarrow \forall x.(x \geq b \rightarrow L[x])} \text{ (IntInd}_{\geq}\text{)}$$

To refute the negated assertion (4), we instantiate the **IntInd**_≥ rule with $L[\sigma]$ being $2 \cdot \text{sum}(\sigma) = \sigma \cdot (\sigma + 1)$ and b set to 0, deriving thus the following induction axiom as an instance of the induction schema of **IntInd**_≥:

$$\begin{aligned} & (2 \cdot \text{sum}(0) = 0 \cdot (0 + 1) \\ & \wedge \forall y \in \mathbb{N}.(y \geq 0 \wedge 2 \cdot \text{sum}(y) = y \cdot (y + 1) \implies 2 \cdot \text{sum}(y + 1) = (y + 1) \cdot ((y + 1) + 1))) \quad (5) \\ & \implies \forall x \in \mathbb{N}.(x \geq 0 \rightarrow 2 \cdot \text{sum}(x) = x \cdot (x + 1)) \end{aligned}$$

Recall that saturation-based provers work with clauses, rather than with arbitrary formulas. Therefore, the induction axiom (5) is clausified and its clausal normal form (CNF) given below is added to the search space, where y is skolemized as σ' :

$$\begin{aligned} & 2 \cdot \text{sum}(0) \neq 0 \cdot (0 + 1) \vee 2 \cdot \text{sum}(\sigma') = \sigma' \cdot (\sigma' + 1) \vee \neg(x \geq 0) \vee 2 \cdot \text{sum}(x) = x \cdot (x + 1) \quad (6) \\ & 2 \cdot \text{sum}(0) \neq 0 \cdot (0 + 1) \vee 2 \cdot \text{sum}(\sigma' + 1) \neq (\sigma' + 1) \cdot ((\sigma' + 1) + 1) \vee \neg(x \geq 0) \vee \\ & \quad \quad \quad 2 \cdot \text{sum}(x) = x \cdot (x + 1) \quad (7) \end{aligned}$$

Optimizing Induction in Saturation. Simply instantiating **IntInd**_≥ and adding the corresponding induction axiom for any clause $\overline{L}[t] \vee C$ in the search space would however be inefficient: considering $\overline{L}[t] \vee C$ just like any other clause in saturation may trigger the application of too many inferences. Therefore, we treat premises $\overline{L}[t] \vee C$ of induction rules differently in order to *guide the saturation algorithm* in two ways.

First, we ensure that an application of **Ind** or **IntInd**_≥ is followed by a binary resolution step in which the conclusion of an induction rule is resolved with (inductive) premise(s). For example, to derive a refutation from (6) and (7), we apply binary resolution on (6) and (7) with (3) and (4), resolving away the last two literals of (6) and (7). Refutation of (4) is then easily derived, by using the axioms (1)–(2) defining **sum** together with arithmetic reasoning over integers. For example, our theorem prover VAMPIRE [9] finds a refutation of (4) in almost no time¹.

Second, induction can be very explosive – i.e., it may generate many consequences of which few lead to refutation. Therefore, in practice, we implement additional requirements on the premises of **Ind** and **IntInd**_≥, with these requirements to be used during saturation. Among others, we use heuristics on whether

¹ Empirical data reported in this paper have been obtained on computers with AMD Epyc 7502 2.5 GHz processors and 1 TB RAM.

```

assume A.size > 0 ∧ A[0] = 0

i := 1;
while i < A.size do
    A[i] := A[i - 1] + i;
    i := i + 1;
    invariant  $\forall j \in \mathbb{Z}. (1 \leq j \leq i - 1 \rightarrow \text{val}_A(j) = \text{val}_A(j - 1) + j)$ 
end

assert  $\forall j \in \mathbb{Z}. (0 \leq j \leq A.size - 1 \rightarrow 2 \cdot \text{val}_A(j) = j \cdot (j + 1))$ 
    
```

Fig. 2. Inductive reasoning with arrays, with $\text{val}_A(j)$ denoting $A[j]$.

the term t must contain a symbol from the conjecture we are trying to prove; whether we apply induction on non-unit clauses; or whether (in the case of integer induction) we allow $L[t]$ to be a comparison or equality literal, and if yes, how many times and on which positions it can contain the term t .

Induction and Theories. We note that the `sum` function and the corresponding assertion of Fig. 1(a) can also be encoded using natural numbers as inductively defined data types. While the resulting encoding of Fig. 1(a) holds over naturals, proving Fig. 1(a) over naturals becomes very complex in practice, as natural numbers do not have built-in arithmetic axioms but rely on term algebra axioms [8]. As a result, when proving Fig. 1(a) over naturals, we are faced with the challenge of proving addition and multiplication properties of naturals, which require induction as well, making efficient proof search challenging. Our work therefore advocates the combination of inductive reasoning with theory-specific inference rules, in the case of Fig. 1(a) this being the application of induction over integers.

Application of induction over integers becomes especially beneficial when proving complex, non-linear arithmetic conjectures. Figure 1(b) shows such a use-case of a function `sum_evsq` that recursively computes the sum of squares of the first n positive even integers. The assertion of Fig. 1(b) is the well-known closed form formula for the sum computed by `sum_evsq`. Proving this assertion, we follow a similar recipe as for Fig. 1(a): instantiate induction inferences over integers with the equality from the assertion, resolve the conclusion of the induction axiom with the literals of the assertion, and then prove the base case and the step case using arithmetic reasoning combined with the definition of `sum_evsq`. Thanks to theory-specific reasoning together with induction, VAMPIRE proves Fig. 1(b) in no time (in less than 1 s).

4 Induction over Arrays

We next describe applications of induction in saturation while proving the functional correctness of array-manipulating programs.

Example of Induction over Arrays. Consider the imperative program of Fig. 2, annotated with pre-condition (**assume**), post-condition (**assert**) and loop invariant (**invariant**).

Given the pre-condition and the invariant, we aim to prove that, upon loop termination, each $A[j]$ will hold the sum of the first j positive integers. Note that the assumed termination of the loop implies the negation of the loop condition: $\neg(i < A.size)$. With this additional formula, the assertion of Fig. 2 clearly holds. Yet, proving it automatically, inductive reasoning is needed.

Induction Rule over Arrays of Integers. We consider the following variant of the IntInd_{\geq} rule, using induction over a finite integer interval:

$$\frac{\overline{L[t] \vee C} \quad t \geq b_1 \quad t \leq b_2}{L[b_1] \wedge \forall x.(b_1 \leq x < b_2 \wedge L[x] \rightarrow L[x+1]) \rightarrow \forall y.(b_1 \leq y \leq b_2 \rightarrow L[y])} (\text{IntInd}_{[\geq]}),$$

where $L[t]$ is a ground literal, and b_1, b_2 are ground terms. We instantiate $\text{IntInd}_{[\geq]}$ based on the negated, skolemized and clausified assertion of Fig. 2; for doing so, we set $L[\sigma]$ to be $2 \cdot \text{val}_A(\sigma) = \sigma \cdot (\sigma + 1)$ and consider b_1 to be 0 and b_2 to be $A.size - 1$. We further clausify the resulting induction axiom; resolve the clausified axiom against the premises of $\text{IntInd}_{[\geq]}$; and finally refute the rest of the literals using the invariant, pre-condition and negated loop condition of Fig. 2 within integer arithmetic.

Note that, unlike in the examples of Fig. 1, one of the bounds of the interval upon which we are applying induction is *symbolic* – an uninterpreted constant. This is a powerful generalization which allows us to reason with arrays regardless of their specific length. In practice, induction over arrays of integers in VAMPIRE proves the assertion of Fig. 2 (using around 1 s of time).

5 Induction over Lists

We finally present our efforts towards proving inductive properties of functional programs, using combination of inductively defined data types. We use two datatypes, natural numbers and lists over natural numbers, denoted respectively by \mathbb{N} and \mathbb{L} . We assume that these datatypes are axiomatised by the *distinctiveness*, *exhaustiveness* and *injectivity* axioms of term algebras [8].

Example of Induction over Lists. Consider the functional program of Fig. 3. We aim to prove the assertion (**assert**) expressing that reversing a list an even number of times results in the same list; doing so, we use an assumption (**assume**) corresponding to an inductive lemma. For proving the assertion of Fig. 3, we translate the function definitions and assumption of Fig. 3 into first-order axioms, negate the assertion of Fig. 3, and clausify the resulting formulas. As a result, the following two clauses are obtained from the negated assertion, respectively introducing Skolem constants σ_1 and σ_2 for n and xs :

$$\text{even}(\sigma_1) \tag{8}$$

$$\text{revN}(\sigma_2, \sigma_1) \neq \sigma_2 \tag{9}$$

<pre> fun even($n : \mathbb{N}$) = match n 0 \Rightarrow \top s(0) \Rightarrow \perp s(s(m)) \Rightarrow even(m) </pre>	<pre> fun rev($xs : \mathbb{L}$) = match xs nil \Rightarrow nil cons(z, zs) \Rightarrow app(rev(zs), cons(z, nil)) </pre>
<pre> fun app($xs : \mathbb{L}, ys : \mathbb{L}$) = match xs nil \Rightarrow ys cons(z, zs) \Rightarrow cons(z, app(zs, ys)) </pre>	<pre> fun revN($xs : \mathbb{L}, n : \mathbb{N}$) = match n 0 \Rightarrow xs s(m) \Rightarrow revN(rev(xs), m) </pre>

assume $\forall xs \in \mathbb{L}, n \in \mathbb{N}. \text{revN}(xs, n) = \text{revN}(\text{rev}(\text{rev}(xs)), n)$
assert $\forall n \in \mathbb{N}, xs \in \mathbb{L}. \text{even}(n) \rightarrow \text{revN}(xs, n) = xs$

Fig. 3. Inductive reasoning with natural numbers and list datatypes.

Induction Rule over Lists. A suitable induction formula that refutes clause (9) is generated in two steps. A formula generated solely from clause (9) may be too strong. Hence, we generate an induction formula that takes clause (8) into account as well. Doing so, we use a generalization of the `Ind` rule that works on an arbitrary number of premises. Namely, we use the following induction rule with two premises:

$$\frac{\overline{L}[t] \vee C \quad \overline{L}'[t] \vee C'}{F \rightarrow \forall x. (L[x] \vee L'[x])} \text{ (Ind')},$$

where $L[t]$ and $L'[t]$ are ground literals, C and C' are clauses, and $F \rightarrow \forall x. (L[x] \vee L'[x])$ is a valid induction schema.

Second, to generate a suitable antecedent for the induction schema (i.e. F), we notice that the recursion used in the definition of `even` suggests an induction principle different from standard structural induction over natural numbers. These insights lead us to generate following induction axiom:

$$\left(\begin{aligned} & (\neg \text{even}(0) \vee \text{revN}(\sigma_2, 0) = \sigma_2) \wedge (\neg \text{even}(s(0)) \vee \text{revN}(\sigma_2, s(0)) = \sigma_2) \\ & \wedge \forall n. ((\neg \text{even}(n) \vee \text{revN}(\sigma_2, n) = \sigma_2) \rightarrow \\ & \quad (\neg \text{even}(s(s(n))) \vee \text{revN}(\sigma_2, s(s(n))) = \sigma_2)) \end{aligned} \right) \\ \rightarrow \forall m. (\neg \text{even}(m) \vee \text{revN}(\sigma_2, m) = \sigma_2)$$

After classifying this axiom and resolving the conclusion literals with the premises (8) and (9), a first-order refutation using the term algebra axioms and the classified function definitions and assumption of Fig. 3 is straightforward; VAMPIRE finds a refutation almost immediately.

Optimizing Induction in Saturation. Note that Fig. 3 uses an auxiliary inductive lemma (assume), in order to prove the assertion of Fig. 3. An additional challenge in automating the proof of the assertion of Fig. 3 comes therefore with the task of generating and proving auxiliary inductive lemmas during saturation.

Proving the lemma of Fig. 3 needs further induction steps; however, the generation of a suitable induction formula is only triggered by an *instance* of the

respective lemma. Since the superposition calculus is optimized to avoid generating clauses unnecessary for first-order reasoning, either (i) we tweak the parameters of superposition such that the generation of an instance of the lemma *is necessary* for first-order reasoning, or (ii) we perform additional sound inferences (on top of superposition and induction inferences) to derive these instances.

Addressing these challenges, we develop different term ordering families (e.g. KBO or LPO), parameterized by various symbol precedences or weight functions; and devise literal selection functions to vary the inferred consequences of a subgoal [5]. As a result, we select different inductive lemmas during saturation. Further, we use function definitions not as axioms but as rewrite rules, in order to ensure that recursively defined functions are expanded/rewritten into their (likely much larger) definitions [6]. With such optimizations at hand, VAMPIRE proves the assertion of Fig. 3 without using the asserted inductive lemma (assume), but by generating the respective inductive lemma of Fig. 3 completely automatically.

6 Conclusions and Outlook

Automated reasoning about system requirements is one of the most active areas of formal methods [2, 10]. Our work addresses recent reasoning demands in the presence of induction, needed for example in proving safety and security requirements over software systems or establishing mathematical conjectures. In particular, we turn saturation-based first-order theorem proving into a powerful workhorse for automating induction. When we integrate induction in saturation, the choice of possibilities to exploit is very large. As such, should one approach fail to bring considerable improvements, one may quickly study and investigate other approaches, allowing thus for further improvements and advancements in mechanizing induction. As saturation-based first-order theorem proving is not yet fully integrated in the tech-chain of ensuring software reliability, we believe automating induction in saturation will bring significant further advances in the theory and practice of both automated reasoning and formal verification.

Acknowledgements. We thank the entire developer team of the VAMPIRE theorem prover, and in particular Daneshvar Amrollahi, Michael Rawson, Giles Reger, and Martin Suda, for valuable discussions. We acknowledge generous funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic Press, New York (1988)
2. Cook, B.: Formal reasoning about the security of Amazon web services. In: CAV, pp. 38–47 (2018)

3. Cruanes, S.: Superposition with structural induction. In: FroCoS (2017)
4. Hajdu, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: CICM (2020)
5. Hajdu, M., Kovács, L., Rawson, M.: Rewriting and inductive reasoning. In: LPAR (2024, to appear)
6. Hajdu, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: FMCAD (2021)
7. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: CADE, pp. 361–377 (2021)
8. Kovács, L., Robillard, S., Voronkov, A.: Coming to terms with quantified reasoning. In: POPL, pp. 260–270 (2017)
9. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: CAV (2013)
10. O’Hearn, P.W.: Continuous reasoning: scaling the impact of formal methods. In: LICS, pp. 13–25 (2018)
11. Passmore, G.O., et al.: The Imandra automated reasoning system (system description). In: IJCAR, pp. 464–471 (2020)
12. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: CADE (2019)
13. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: VMCAI, pp. 80–98 (2015)
14. Wand, D.: Superposition: types and induction. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2017). <https://tel.archives-ouvertes.fr/tel-01592497>

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

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





Stepping Stones in the TPTP World

Geoff Sutcliffe^(✉) 

University of Miami, Miami, USA

geoff@cs.miami.edu

Abstract. The TPTP World is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. There are key components that help make the TPTP World a success: the TPTP problem library was first released in 1993, the CADE ATP System Competition (CASC) was conceived after CADE-12 in 1994, problem difficulty ratings were added in 1997, the current TPTP language was adopted in 2003, the SZS ontologies were specified in 2004, the TSTP solution library was built starting around 2005, the Specialist Problem Classes (SPCs) have been used to classify problems since 2010, the SystemOnTPTP service has been offered from 2011, the StarExec service was started in 2013, and a world of TPTP users have helped all along. This paper reviews these stepping stones in the development of the TPTP World.

1 Introduction

The TPTP World [38, 41] (once gently referred to as the “TPTP Jungle” [3]) is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. Salient components of the TPTP World are the TPTP languages [47], the TPTP problem library [37], the TSTP solution library [38], the SZS ontologies [36], the Specialist Problem Classes (SPCs) and problem difficulty ratings [51], SystemOnTPTP [33] and StarExec [32], and the CADE ATP System Competition (CASC) [40]. There are dependencies between these parts of the TPTP World, as shown in Fig. 1, forming a series of “stepping stones” from TPTP standards to happy users, described in the following sections of this paper . . .

- The TPTP language (Sect. 2) is used for writing problems in the TPTP problem library and the TSTP solution library.
- The TPTP problem library (Sect. 3) is the central collection of test problems.
- The TSTP solution library (Sect. 4) is the central collection of solutions to the TPTP library problems.
- The SZS ontologies (Sect. 5) are used to specify properties of problems and solutions.
- The SPCs (Sect. 6) that classify problems are based on the language form and SZS ontology values.

- The TPTP problems’ difficulty ratings (Sect. 7) are computed wrt each SPC, using data from the TSTP solution library.
- SystemOnTPTP (Sect. 8) provides online access to ATP systems and tools.
- The StarExec computers (Sect. 8) are used to build the TSTP, and to run CASC.
- The CADE ATP System Competition (Sect. 9) is the annual evaluation of ATP systems - the world championship for such systems.
- Users of the TPTP World (Sect. 10) provide problems for the TPTP problem library, and ATP systems/tools for SystemOnTPTP and StarExec.

There is a cycle of dependencies from the TPTP problem library, to the TSTP solution library, to the problem difficulty ratings, and back to the TPTP problem library. This cycle means that building these components, particularly building releases of the TPTP problem library, requires iteration until stability is reached.

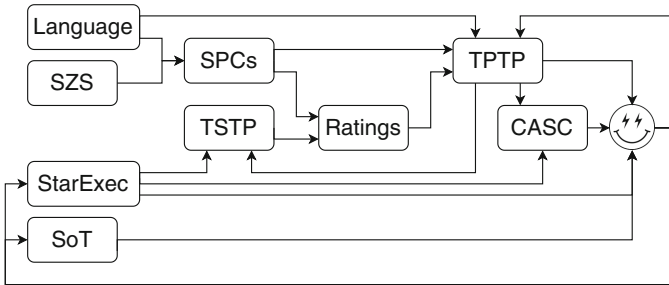


Fig. 1. Dependencies between the Stepping Stones

Various parts of the TPTP World have been deployed in a range of applications, in both academia and industry. Since the first release of the TPTP problem library in 1993, many researchers have used the TPTP World as an appropriate and convenient basis for ATP system research and development. The web page www.tptp.org provides access to all components.

2 The TPTP Language

The TPTP language [42] is one of the keys to the success of the TPTP World. The TPTP language is used for writing both problems and solutions, which enables convenient communication between ATP systems and tools. Originally the TPTP World supported only first-order clause normal form (CNF) [50]. Over the years full first-order form (FOF) [37], typed first-order form (TFF) [2, 46], typed extended first-order form (TXF) [44], typed higher-order form (THF) [12, 43], and non-classical forms (NTF) [30] have been added. The standardisation received a significant technical boost in 2006 when the BNF definition was revised

so that all the language forms are quite precisely specified.¹ As a result a parser can be generated using `lex/yacc` [54]. A general principle of the TPTP language is: “we provide the syntax, you provide the semantics”. As such, there is no a priori commitment to any semantics for each of the language forms, although in almost all cases the intended logic and semantics are well known.

Problems and solutions are built from *annotated formulae* of the form ...
language(name, role, formula, source, useful_info)

The *languages* supported are `cnf` (clause normal form), `fof` (first-order form), `tff` (typed first-order form), and `thf` (typed higher-order form). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula. In a *formula*, terms and atoms follow Prolog conventions – functions and predicates start with a lowercase letter or are ‘single quoted’, and variables start with an uppercase letter. The language also supports interpreted symbols that either start with a \$, e.g., the truth constants `$true` and `$false`, or are composed of non-alphabetic characters, e.g., integer/rational/real numbers such as 27, 43/92, -99.66. The logical connectives in the TPTP language are `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<->`, for the mathematical connectives \forall , \exists , \neg , \vee , \wedge , \Rightarrow , \Leftarrow , \Leftrightarrow , and \oplus respectively. Equality and inequality are expressed as the infix operators `=` and `!=`. The *source* and *useful_info* are optional. Figure 2 shows an example with typed higher-order annotated formulae.

3 The TPTP Problem Library

The first inklings of the TPTP World emerged at CADE-10 in Kaiserslautern, Germany, in 1990, as a collaboration between Geoff Sutcliffe from the University of Western Australia (James Cook University from 1993), and Christian Suttner from the Technische Universität München. At that time a large number of test problems had accumulated in the ATP community, in both hard-copy [4, 14, 17, 22, 25, 56] and electronic form [13, 29].² We observed that ATP system developers were testing their systems, and publishing results, based on small numbers of selected test problems. At the same time some good ideas were seen to be abandoned because they had been tested on inappropriate selections of test problems. These observations motivated us to start collecting ATP test problems into what became the TPTP problem library. A comprehensive library of test problems is necessary for meaningful system evaluations, meaningful system comparisons, repeatability of testing, and the production of statistically significant results. The goal was to support testing and evaluation of ATP systems, to help ensure that performance results accurately reflect the capabilities of the ATP systems being considered.

¹ But note that the BNF “syntax” is not completely strict – the BNF uses an extended BNF form that relegates details to “strict” rules that do not have to be checked by a parser.

² To my knowledge, the first circulation of test problems was by Larry Wos in the late sixties.

```

%-----
thf(beverage_decl,type, beverage: $tType ).
thf(syrup_decl,type, syrup: $tType ).
thf(coffee_type,type, coffee: beverage ).
thf(mix_type,type, mix: beverage > syrup > beverage ).
thf(heat_type,type, heat: beverage > beverage ).
thf(heated_mix_type,type, heated_mix: beverage > syrup > beverage ).
thf(hot_type,type, hot: beverage > $o ).

thf(heated_mix,axiom,
  ( heated_mix
    = ( ~ [B: beverage,S: syrup] : ( heat @ ( mix @ B @ S ) ) ) ).

thf(hot_mixture,axiom,
  ! [B: beverage,S: syrup] : ( hot @ ( heated_mix @ B @ S ) ) ).

thf(heated_coffee_mix,axiom,
  ! [S: syrup] : ( ( heated_mix @ coffee @ S ) = coffee ) ).

thf(hot_coffee,conjecture,
  ? [Mixture: syrup > beverage] :
  ! [S: syrup] :
  ( ( ( Mixture @ S ) = coffee )
    & ( hot @ ( Mixture @ S ) ) ) ).
%-----

```

Fig. 2. THF annotated formulae

Releases of the TPTP problem library are identified by numbers in the form *vVersion.Edition.Patch*. The *Version* enumerates major new releases of the TPTP in which important new features have been added. The *Edition* is incremented each time new problems are added to the current version. The *Patch* level is incremented each time errors found in the current edition are corrected. The first release of the TPTP problem library, v1.0.0 was made on Friday 12th November 1993.

The problems in the library are classified into *domains* that reflect a natural hierarchy, based mainly on the Dewey Decimal Classification and the Mathematics Subject Classification. Seven main fields are defined: logic, mathematics, computer science, science & engineering, social sciences, arts & humanities, and other. Each field is subdivided into domains, each identified by a three-letter mnemonic, e.g., the social science field has three domains: Social Choice Theory (SCT), Management (MGT), and Geography (GEG).

Table 1 lists the versions of the TPTP up to v9.0.0, with the new feature added, the number of problem domains, and the number of problems.³ The number of domains indicates the diversity of the problems, while the number of problems indicates the size of the library. The attentive reader might note that many releases have been made in July-September. This is because the CADE ATP System Competition (CASC - see Sect. 9), has an influence on the release cycle of the TPTP.

³ The data for v9.0.0 is an estimate, because this paper was written before the release was finalised.

Table 1. Overview of TPTP releases

Release	Date	Changes	D's	P's
v1.0.0	12/11/93	First public release, only CNF [50]	23	2295
v2.0.0	05/06/97	FOF [37] and ratings (Sect. 7)	28	3277
v3.0.0	11/11/04	New TPTP language [47]	32	7267
v4.0.0	04/07/09	TH0 (monomorphic typed higher-order) [43]	41	16512
v5.0.0	16/09/10	TF0 (monomorphic typed first-order) [46]	45	18480
v6.0.0	21/09/13	TF1 (polymorphic typed first-order) [2]	48	20306
v7.0.0	24/07/17	TH1 (polymorphic typed higher-order) [12]	53	21851
v8.0.0	19/04/22	TXF (typed extended first-order) [44]	54	24785
v9.0.0	??/07/24	NTF (non-classical typed first-order) [30]	55	25598

Each TPTP problem file has three parts: a header, optional includes, and annotated formulae. The header section contains information for users, formatted as comments in four parts, as shown in Fig. 3. The first part identifies and describes the problem, the second part provides information about occurrences of the problem in the literature and elsewhere, the third part provides semantic and syntactic characteristics of the problem, and the last part contains comments and bugfix information. The header fields are self-explanatory, but of particular interest are the **Status** field that is explained in Sect. 5, the **Rating** field that is explained in Sect. 7, and the **SPC** field that is explained in Sect. 6. The include section is optional, and if used contains `include` directives for axiom files, which in turn have the same three-part format as problem files; see Fig. 3 for an example. Inclusion avoids the need for duplication of the formulae in commonly used axiomatizations. The annotated formulae are described in Sect. 2, and Fig. 2 provides an example.

4 The TSTP Solution Library

The complement to the TPTP problem library is the TSTP solution library [35, 38]. The TSTP is built by running all the ATP systems that are available in the TPTP World on all the problems in the TPTP problem library. The TSTP started being built around 2005, using solutions provided by individual system developers. From 2010 to 2013 the TSTP was generated on a small NSF funded⁴ cluster at the University of Miami. Since 2014 the ATP systems have been run on StarExec (see Sect. 8), initially on the StarExec Iowa cluster, and since 2018 on the StarExec Miami cluster. StarExec has provided stable platforms that produce reliably consistent and comparable data in the TSTP. At the time of writing this paper the TSTP contained the results of running 87 ATP systems and system variants on all the problems that each system could attempt (therefore, e.g.,

⁴ NSF Award 0957438.

```

%-----
% File      : DAT016_1 : TPTP v8.2.0. Bugfixed v5.1.0.
% Domain    : Data Structures
% Problem   : Some element is 53
% Version   : [PW06] axioms.
% English   : Show that some element of the array has the value 53.

% Refs     : [PW06] Prevosto & Waldmann (2006), SPASS+T
%           : [Wal10] Waldmann (2010), Email to Geoff Sutcliffe
% Source    : [Wal10]
% Names     : (40) [PW06]

% Status    : Theorem
% Rating    : 0.25 v8.2.0, 0.12 v7.5.0, 0.30 v7.4.0, 0.12 v7.3.0, etc.
% Syntax    : Number of formulae      : 6 ( 1 unt; 3 typ; 0 def)
%           : Number of atoms         : 12 ( 5 equ)
%           : Maximal formula atoms   : 4 ( 2 avg)
%           : Number of connectives   : 4 ( 0 <; 1 |; 1 &)
%           :                         ( 0 <=>; 2 =>; 0 <; 0 <?>)
%           : Maximal formula depth   : 6 ( 5 avg)
%           : Maximal term depth      : 3 ( 1 avg)
%           : Number of FOOLs         : 5 ( 5 fml; 0 var)
%           : Number arithmetic       : 16 ( 2 atm; 2 fun; 5 num; 7 var)
%           : Number of types         : 2 ( 1 usr; 1 ari)
%           : Number of type conns    : 5 ( 2 >; 3 *; 0 +; 0 <<)
%           : Number of predicates    : 3 ( 1 usr; 0 prp; 2-2 aty)
%           : Number of functors      : 9 ( 2 usr; 5 con; 0-3 aty)
%           : Number of variables     : 10 ( 9 !; 1 ?; 10 :)
% SPC       : TFO_THM_EQU_ARI

% Comments  : The array contains integers.
% Bugfixes  : v5.1.0 - Fixed conjecture
%-----
%----Includes axioms for arrays
include('Axioms/DAT001_0.ax').
%-----

```

Fig. 3. Header of problem DAT016_1.

systems that do model finding for FOF are not run on THF problems). This produced 1091026 runs, of which 432718 (39.6%) solved the problem. One use of the TSTP is for computing the TPTP problem difficulty ratings (see Sect. 7).

TSTP solution files have a header section and annotated formulae. The header section has four parts, as shown in Fig. 4: the first part identifies the ATP system, the problem, and the system’s runtime parameters; the second part provides information about the hardware, operating system, and resource limits; the third part provides the SZS result and output values (see Sect. 5), and syntactic characteristics of the solution; the last part contains comments. The solution follows in annotated formulae.

For derivations, where formulae are derived from parent formulae, e.g., in proofs, refutations, etc., the *source* fields of the annotated formulae are used to capture parent-derived formulae relationships in the derivation DAG. This includes the source of the formula – either the problem file or an inference. Inference data includes the name of the inference rule used, the semantic relationship between the parents and the derived formula as an SZS ontology value (see Sect. 5), and a list of the parent annotated formulae names. Figure 5 shows an example refutation [slightly modified, type declarations omitted] from the E system [27] for the problem formulae in Fig. 2, and Fig. 4 shows the corresponding header fields.

```

%-----
% File      : E--3.0.04
% Problem   : Coffee
% Transfm   : none
% Format     : tptp:raw
% Command   : run_E %s %d THM

% Computer  : quokka.cs.miami.edu
% Model     : x86_64 x86_64
% CPU       : Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz
% Memory    : 64222MB
% OS        : Linux 3.10.0-1160.36.2.el7.x86_64
% CPULimit  : 30s
% WCLimit   : 0s
% DateTime  : Tue Feb 13 13:29:34 EST 2024

% Result    : Theorem 0.00s 0.08s
% Output    : Refutation 0.00s
% Verified  :
% SZS Type  : Refutation
%           : Derivation depth      : 5
%           : Number of leaves      : 10
% Syntax    : Number of formulae    : 19 ( 8 unt; 7 typ; 0 def)
%           : Number of atoms       : 16 ( 7 equ; 0 con)
%           : Maximal formula atoms : 2 ( 1 avg)
%           : Number of connectives : 42 ( 6 ~; 2 |; 2 &; 32 @)
%           :                       ( 0 <=>; 0 =>; 0 <=; 0 <~>)
%           : Maximal formula depth : 7 ( 5 avg)
%           : Number of types       : 3 ( 2 usr)
%           : Number of type conns  : 11 ( 11 >; 0 *; 0 +; 0 <<)
%           : Number of symbols     : 7 ( 5 usr; 2 con; 0-2 aty)
%           : Number of variables   : 15 ( 0 ~ 13 !; 2 ?; 15 :)

% Comments :
%-----

```

Fig. 4. Example derivation header

For interpretations (typically models) the annotated formulae are used to describe the domains and symbol mappings of Tarskian interpretations, or the formulae that induce Herbrand models. A TPTP format for interpretations with finite domains was previously defined [47], and has served the ATP community adequately for almost 20 years. Recently the need to represent interpretations with infinite domains, and Kripke interpretations, has led to the development of a new TPTP format that supports these interpretations [49]. Figure 6 shows the problem formulae and a model that uses integers as the domain. Please read [49] for lots more details!

Resource Limits: A common question, often based on mistaken beliefs, is whether the resource limits used should be increased to find more solutions. Analysis shows that increasing resource limits does not significantly affect which problems are solved by an ATP system. Figure 7 illustrates this point; it plots the CPU times taken by several contemporary ATP systems to solve the TPTP problems for the FOF_THM_RFO_* SPCs, in increasing order of time taken. The data was taken from the TSTP, i.e., using the StarExec Miami computers. The relevant feature of these plots is that each system has a point at which the time taken to find solutions starts to increase dramatically. This point is called the system’s Peter Principle [23] Point (PPP) – it is the point at which the system has reached its level of incompetence. Evidently a linear increase in the computational resources beyond the PPP would not lead to the solution of significantly

```

%-----
thf(hot_coffee,conjecture,
  ? [X3: syrup > beverage] :
  ! [X2: syrup] :
  ( ( X3 @ X2 ) = coffee ) & ( hot @ ( X3 @ X2 ) ) ,
  file('Coffee.p',hot_coffee) ).

thf(heated_coffee_mix,axiom,
  ! [X2: syrup] :
  ( ( heated_mix @ coffee @ X2 ) = coffee ) ,
  file('Coffee.p',heated_coffee_mix) ).

thf(hot_mixture,axiom,
  ! [X1: beverage,X2: syrup] : ( hot @ ( heated_mix @ X1 @ X2 ) ) ,
  file('Coffee.p',hot_mixture) ).

thf(c_0_3,negated_conjecture,
  ~ ? [X3: syrup > beverage] :
  ! [X2: syrup] :
  ( ( X3 @ X2 ) = coffee ) & ( hot @ ( X3 @ X2 ) ) ,
  inference(assume_negation,[status(cth)],[hot_coffee]) ).

thf(c_0_4,negated_conjecture,
  ! [X16: syrup > beverage] :
  ( ( X16 @ ( esk1_1 @ X16 ) ) != coffee )
  | ~ ( hot @ ( X16 @ ( esk2_1 @ X16 ) ) ) ,
  inference(fof_nnf,[status(thm)],[inference(skolemize,[status(esa)],
[inference(variable_rename,[status(thm)],[inference(shift_quantors,
[status(thm)],[inference(fof_nnf,[status(thm)],[c_0_3]))]))]) ) ).

thf(c_0_10,negated_conjecture,
  ~ ( hot @ coffee ) ,
  inference(cn,[status(thm)],[inference(rw,[status(thm)],
[inference(spm,[status(thm)],[c_0_4,heated_coffee_mix]),
heated_coffee_mix])) ) ).

thf(c_0_11,plain,
  $false,
  inference(sr,[status(thm)],[inference(spm,[status(thm)],
[hot_mixture,heated_coffee_mix]),c_0_10]),[proof] ).
%-----

```

Fig. 5. Example derivation formulae

more problems. The PPP thus defines a realistic computational resource limit for the system, and if enough CPU time is allowed for an ATP system to reach its PPP, a usefully accurate measure of what problems it can solve is achieved. The performance data in the TSTP is produced with adequate resource limits.

5 The SZS Ontologies

The SZS ontologies [36] (named “SZS” after the authors of the first presentation of the ontologies [52]) provide values to specify the logical status of problems and solutions, and to describe logical data. The Success ontology provides values for the logical status of a conjecture with respect to a set of axioms, e.g., a TPTP problem whose conjecture is a logical consequence of the axioms is tagged as a **Theorem** (as in Fig. 3), and a model finder that establishes that

```

%---- Problem formulae -----
tff(person_type,type,          person: $tType ).
tff(bob_decl,type,            bob: person ).
tff(child_of_decl,type,       child_of: person > person ).
tff(is_descendant_decl,type,  is_descendant: ( person * person ) > $o ).

tff(descendents_different,axiom,
    ! [A: person,D: person] :
      ( is_descendant(A,D) => ( A != D ) ) ).

tff(descendent_transitive,axiom,
    ! [A: person,C: person,G: person] :
      ( ( is_descendant(A,C) & is_descendant(C,G) )
        => is_descendant(A,G) ) ).

tff(child_is_descendant,axiom,
    ! [P: person] : is_descendant(P,child_of(P)) ).

tff(all_have_child,axiom,
    ! [P: person] : ? [C: person] : C = child_of(P) ).
%-----
%---- Model -----
tff(person_type,type,          person: $tType ).
tff(bob_decl,type,            bob: person ).
tff(child_of_decl,type,       child_of: person > person ).
tff(is_descendant_decl,type,  is_descendant: ( person * person ) > $o ).

tff(int2person_decl,type,     int2person: $int > person ).

tff(people,interpretation,
%----Domain for type person is the integers
  ( ( ! [P: person] : ? [I: $int] : int2person(I) = P
%----The type promoter is a bijection (injective and surjective)
    & ! [I1: $int,I2: $int] :
      ( int2person(I1) = int2person(I2) => I1 = I2 ) )
%----Mapping people to integers. Note that Bob's ancestors will be
%----interpreted as negative integers.
    & ( bob = int2person(0)
%----Interpretation of descendancy
    & ! [A: $int,D: $int] :
      ( is_descendant(int2person(A),int2person(D)) <=> $less(A,D) ) ) ).
%-----

```

Fig. 6. Example infinite model

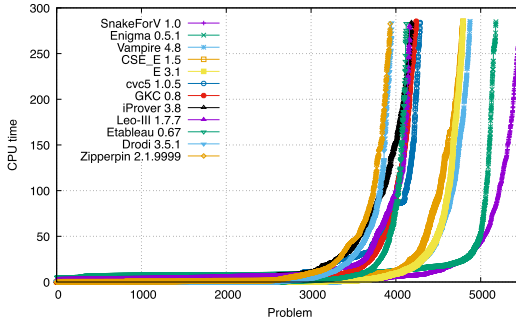


Fig. 7. CPU times for FOF_THM_RFO_*

a set of axioms (with no conjecture) is consistent should report **Satisfiable**. The Success ontology is also used to specify the semantic relationship between the parent “axioms” and inferred “conjecture” of an inference, as done in the TPTP format for derivations (see Sect. 4). The NoSuccess ontology provides reasons why an ATP system/tool has failed, e.g., an ATP system might report **Timeout**. The Dataform ontology provides values for describing logical data, as might be output from an ATP system/tool, e.g., a model finder might output a **FiniteModel**. Figure 8 shows some of the salient nodes of the ontologies. Their expanded names and their (abbreviated) definitions are listed in Fig. 9.

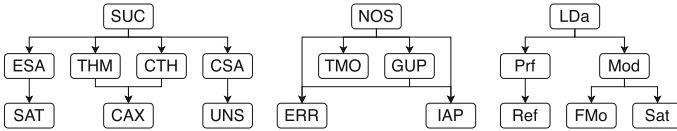


Fig. 8. Extract of the SZS ontologies

The SZS standard also specifies the precise way in which the ontology values should be presented in ATP system output, in order to facilitate easy processing. For example, if an ATP system has established that a conjecture is not a theorem of the axioms, by finding a finite countermodel of the axioms and negated conjecture, the SZS format output would be (see Sect. 4 for the format of the annotated formulae):

```

% SZS status CounterSatisfiable
% SZS output start FiniteModel
... annotated formulae for the finite model
% SZS output end FiniteModel

```

6 Specialist Problem Classes

The TPTP problem library is divided into Specialist Problem Classes (SPCs) – classes of problems with the same recognizable logical, language, and syntactic characteristics that make the problems in each SPC homogeneous wrt ATP systems. Evaluation of ATP systems within SPCs makes it possible to say which systems work well for what types of problems. The appropriate level of subdivision for SPCs is that at which less subdivision would merge SPCs for which ATP systems have distinguishable behaviour, and at which further subdivision would unnecessarily split SPCs for which ATP systems have reasonably homogeneous behaviour. Empirically, homogeneity is ensured by examining the patterns of system performance across the problems in each SPC. The SPCs for essentially propositional problems were motivated by the observation that SPASS [55] performed differently on the ALC problems in the SYN domain of the TPTP. A data-driven test for homogeneity is also possible [6].

SUC Success	Data has been processed successfully.
ESA Equisatisfiable	There exists a model of the axioms iff there exists a model of the conjecture.
SAT Satisfiable	Some interpretations are models of the axioms.
THM Theorem	All models of the axioms are models of the conjecture.
CTH CounterTheorem	All models of the axioms are models of the negated conjecture.
CAX ContradictoryAxioms	No interpretations are models of the axioms.
CSA CounterSatisfiable	Some models of the axioms are models of the negated conjecture.
UNS Unsatisfiable	All interpretations are models of the negated set of axioms.
NOS NoSuccess	Data has not been processed successfully.
ERR Error	Stopped due to an error.
TMO Timeout	Stopped because a time limit ran out.
GUP GaveUp	Gave up of its own accord.
IAP Inappropriate	Cannot process this type of data.
LDa LogicalData	Logical data.
Prf Proof	A proof.
Ref Refutation	A refutation (ending with <i>false</i>).
Mod Model	A model.
FMo FiniteModel	A model with a finite domain.
Sat Saturation	A saturated set of formulae.

Fig. 9. SZS ontology values

The characteristics currently used to define the SPCs in the TPTP are shown in Fig. 10. Using these characteristics 223 SPCs are defined in TPTP v8.2.0. For example, the SPC `TF0_THM_NEQ_ARI` contains typed monomorphic first-order theorems that have no equality but include arithmetic. The header section of each problem in the TPTP problem library (see Sect. 3) includes its SPC. The SPCs are used when computing the TPTP problems difficulty ratings (see Sect. 7).

7 Problem Difficulty Ratings

Each TPTP problem has a difficulty rating that provides a well-defined measure of how difficult the problem is for current ATP systems [51]. The ratings are based on performance data in the TSTP solution library (see Sect. 4), and are updated in each TPTP edition.

The TPTP tags problems that are designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased* (this includes the `SYN000` problems, which are designed for testing ATP systems' parsers). The biased problems are excluded from the calculations. Rating is then done separately for each SPC (see Sect. 6).

- TPTP language:

CNF – Clause Normal Form	FOF – First-Order Form
TF0/1 – Typed First-order	Monomorphic/Polymorphic
TX0/1 – Typed eXtended f-o	Monomorphic/Polymorphic
TH0/1 – Typed Higher-order	Monomorphic/Polymorphic
NX0/1 – Non-classical TX0/1	NHO/1 – Non-classical TH0/1
- SZS status:

THM – Theorem	CSA – CounterSATisfiabIe
CAX – Contradictory AXioms	
UNS – UNSatisfiable	SAT – SATisfiabIe
UNK – UNKown	OPN – OPeN
- Order (for CNF and FOF):
 - PRP – PRoPositional
 - EPR – Effectively PRoPositional (known to be reducible to PRP)
 - RFO – Real First-Order (not known to be reducible to PRP)
- Equality:

NEQ – No EQuality	EQU – EQuality (some or pure)
SEQ – Some (not pure) EQuality	PEQ – Pure EQuality
UEQ – Unit EQuality CNF	NUE – Non-Unit Equality CNF
- Hornness (for CNF):

HRN – HoRN	NHN – Non-HorN
------------	----------------
- Arithmetic (for T* languages):

ARI – ARITHmetic	NAR – No ARITHmetic
------------------	---------------------

Fig. 10. SPC characteristics

First, a partial order between systems is determined according to whether or not a system solves a strict superset of the problems solved by another system. If a strict superset is solved, the first system is said to *subsume* the second. The union of the problems solved by the non-subsumed systems defines the state-of-the-art – all the problems that are solved by any system. The fraction of non-subsumed systems that fail on a problem is the difficulty rating for the problem: problems that are solved by all of the non-subsumed systems get a rating of 0.00 (easy); problems that are solved by some of the non-subsumed systems get a rating between 0.00 and 1.00 (difficult); problems that are solved by none of the non-subsumed systems get a rating of 1.00 (unsolved). As additional output, the systems are given a rating for each SPC – the fraction of difficult problems they can solve.

The TPTP difficulty ratings provide a way to assess progress in the field – as problems that are unchanged are not actually getting easier, decreases in their difficulty ratings are evidence of progress in ATP systems. Figures 11 and 12 plot the average difficulty ratings overall and for each of the four language forms in the TPTP World (after some sensible data cleaning). Figure 11 is taken from [41], published in 2017. It plots the average ratings for the 14527 problems that had been unchanged and whose ratings had not been stuck at 0.00 or 1.00, from

TPTP v5.0.0 that was released in 2010 to v6.4.0 that was released in 2016. Figure 12 is taken from [45], published in 2024. It plots the average ratings for the 16236 problems that had been unchanged and whose ratings had not been stuck at 0.00 or 1.00, from TPTP v6.3.0 that was released in 2016 to v8.2.0 that was released in 2023. The two figures’ plots dovetail quite well, which gives confidence that they really are comparable (there are some minor differences caused by slightly different data cleaning and by recent refinements to the difficulty rating calculations). The older plots show a quite clear downward trend both overall and for the four language forms, while the new plots do not. Possible reasons are discussed in [45].

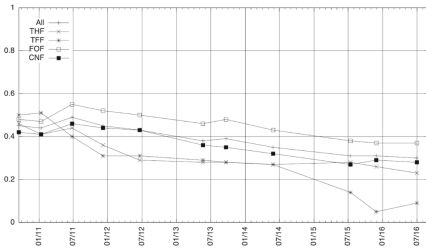


Fig. 11. Ratings from v5.0.0 to v6.4.0

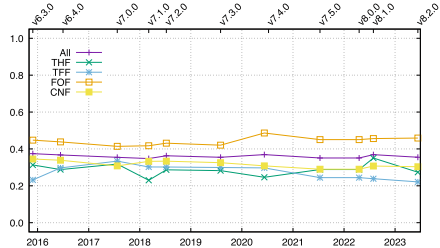


Fig. 12. Ratings from v6.3.0 to v8.2.0

8 SystemOnTPTP and StarExec

Some of the early users of the TPTP problem library (see Sect. 3) were working in disciplines other than computer science, e.g. (with a few exemplar references), mathematics [16, 26], logic [8, 11], management [20, 21], planning [28]. These users often selected the Otter system [15] for their experiments, as it was readily available and easy enough to install. As the TPTP World evolved it was clear that more powerful ATP systems were available, especially evident in the CADE ATP System Competition (see Sect. 9). However, these more powerful systems were often not as easy to obtain and install. In order to make the use of ATP easier for non-geek users, an NSF grant was obtained⁵ to build the SystemOnTPTP online service, which provides hardware and a web interface for users to submit their problems to most recent versions of a wide range of ATP systems. SystemOnTPTP can provide recommendations for ATP systems that might be most likely to solve a problem, based on the SPC of the user’s problem and the system ratings for the SPC (see Sects. 6 and 7). SystemOnTPTP can also run ATP systems (e.g., the recommended systems) in competition parallel [48]. The core SystemOnTPTP service is supported by (i) the SystemB4TPTP service that provides tools to prepare problems for submission to ATP systems, e.g.,

⁵ NSF Award 1405674.

axiom selection [9], type checking [12], a scripting language [39], and (ii) the SystemOnTSTP service that provides tools for analysing solutions output from SystemOnTPTP, e.g., interactive viewing of derivations [53], proof checking [34], identification of interesting lemmas in a proof [24]. While many users enjoy interactive use of the services in a web browser, it is also easy to use the services programmatically – in fact this is where most of the use comes from. In 2023 there were 887287 requests serviced (an average of one every 36 s), from 286 countries. One heavy programmatic user is the Sledgehammer component of the interactive theorem prover Isabelle [19].

The TPTP problem library was motivated (see Sect. 3) by the need to provide support for meaningful ATP system evaluation. This need was also (or became) evident in other logic solver communities, e.g., SAT [10] and SMT [5]. For many years testing of logic solvers was done on individual developer’s computers. In 2010 a proposal for centralised hardware and software support was developed, and in 2011 a \$2.11 million NSF grant⁶ was obtained. This grant led to the development and availability of StarExec Iowa [32] in 2012, and a subsequent \$1.00 million grant⁷ in 2017 expanded StarExec to Miami. StarExec has been central to much progress in logic solvers over the last 10 years, supporting 16 logic solver communities, used for running many annual competitions [1], and supporting many many users.

It was recently announced that StarExec Iowa will be decommissioned. The maintainer of StarExec Iowa explained that “the plan is to operate StarExec as usual for competitions Summer 2024 and Summer 2025, and then put the system into a read-only mode for one year (Summer 2025 to Summer 2026)”. While StarExec Miami will continue to operate while funding is available, it will not be able to support the large number of logic solver communities that use the larger StarExec Iowa cluster. In the long run it will be necessary for StarExec users to transition to new environments, and several plans are (at the time of writing) being discussed. One effort, funded by an Amazon Research Award⁸, is to containerise StarExec and ATP systems so they will run in a Kubernetes framework on Amazon Web Services [7]. This will allow communities and individual users to run their own StarExec.

9 The CADE ATP System Competition

The CADE ATP System Competition (CASC) [40] is the annual evaluation of fully automatic, classical logic, ATP systems - the world championship for such systems. CASC is held at each CADE (the International Conference on Automated Deduction) and IJCAR (the International Joint Conference on Automated Reasoning) conference – the major forums for the presentation of new

⁶ NSF Awards 1058748 and 1058925, led by Aaron Stump and Cesare Tinelli at the University of Iowa.

⁷ NSF Award 1730419.

⁸ Amazon Research Award “Automated Theorem Proving Community Infrastructure in the AWS Cloud”, www.amazon.science/research-awards/recipient/geoffrey-sutcliffe.

research in all aspects of automated deduction. One purpose of CASC is to provide a public evaluation of the relative capabilities of ATP systems. Additionally, CASC aims to stimulate ATP research, motivate development and implementation of robust ATP systems that can be easily and usefully deployed in applications, provide an inspiring environment for personal interaction between ATP researchers, and expose ATP systems within and beyond the ATP community. Over the years CASC has been a catalyst for impressive improvements in ATP, stimulating both theoretical and implementation advances [18]. It has provided a forum at which empirically successful implementation efforts are acknowledged and applauded, and at the same time provides a focused meeting at which novice and experienced developers exchange ideas and techniques. The first CASC was held at CADE-13 in 1996, at DIMACS in Rutgers University, USA. The CASC web site provides access to all the details: www.tptp.org/CASC. Of particular interest for this IJCAR is that CASC was conceived 30 years ago in 1994 after CADE-12 in Nancy, when Christian Suttner and I were sitting on a bench under a tree in Parc de la Pépinière, burning time before our train departures.

CASC is run in divisions according to problem and system characteristics, in a coarse version of the SPCs (see Sect. 6). Problems for CASC are taken from the TPTP problem library (see Sect. 3), and some other sources. The TPTP problem ratings (see Sect. 7) are used to select appropriately difficult problems from the TPTP problem library. The systems are ranked according to the number of problems solved with an acceptable proof/model output. Ties are broken according to the average time over problems solved.

The design and organization of CASC has evolved over the years to a sophisticated state. In the years from CASC-26 in 2017 to CASC-29 in 2023 the competition stayed quite stable, but each year the various divisions, evaluations, etc., were optimized (as was also the case in prior years when there were also larger changes in the competition design). The changes in the divisions reflect the changing demands and interest in different types of ATP problems, and decisions made for CASC (in the context of the TPTP World) have had an influence on the directions of development in ATP. Over the years 11 divisions have been run ...

- The Clause Normal Form (CNF) division from CASC-13 in 1996 to CASC-23 in 2011.
- The Satisfiability (SAT) division from CASC-14 in 1997 to CASC-22 in 2009.
- The First-order Form (FOF) division from CASC-15 in 1998, ongoing.
- The Effectively Propositional (EPR) division from CASC-JC in 2001 to CASC-27 in 2019.
- The First-order Non-theorem (FNT) division from CASC-21 in 2007 to CASC-29 in 2023.
- The Large Theory Batch (LTB) division from CASC-J4 in 2008 to CASC-J11 in 2022.
- The Typed Higher-order (THF) division from CASC-J5 in 2010, ongoing.
- The Typed First-order with Arithmetic (TFA) division from CASC-23 in 2011, ongoing.

- The Typed First-order Non-theorem (TFN) division from CASC-25 in 2015 to CASC-J8 in 2016, revived in CASC-29 in 2003, ongoing.
- The Sledgehammer (SLH) division from CASC-26 in 2017, ongoing.
- The I Challenge You (ICU) division introduced for CASC-J12 in 2024.

In the 20 CASCs so far 111 distinct ATP systems have been entered. For each CASC the division winners of the previous CASC are automatically entered to provide benchmarks against which progress can be judged. Some systems have emerged as dominant in some of the divisions, with Vampire being a well-known example. The strengths of these systems stem from four main areas: solid theoretical foundations, significant implementation efforts (in terms of coding and data structures), extensive testing and tuning, and an understanding of how to optimize for CASC.

10 TPTP World Users

Over the years the TPTP World has provided a platform upon which ATP users have presented their needs to ATP system developers, who have then adapted their ATP systems to the users' needs. The interplay between the TPTP problem library (see Sect. 3) and the CADE ATP System Competition (see Sect. 9) has been particularly effective as a conduit for ATP users to provide samples of their problems to ATP system developers. Users' problems that are contributed to the TPTP are eligible for use in CASC. The problems are then exposed to ATP system developers, who improve their systems' performances on the problems, in order to perform well in CASC. This completes a cycle that provides the users with more effective tools for solving their problems.

Many people have contributed to the TPTP World, with problems, software, advice, expertise, and enthusiasm. I am grateful to them all⁹ and here are just a few who have made salient contributions (ordered roughly by date of the contributions mentioned):

- Christian Suttner - The TPTP problem library and CASC.
- Jeff Pelletier - Early support and thoughtful advice, problem contributions.
- Stephan Schulz - Technical expertise.
- Andrei Voronkov & Vampire team - Enthusiasm and constructive feedback.
- Allen van Gelder - The TPTP language BNF and parser.
- Adam Pease and Josef Urban - Large theory problems.
- Christoph Benzüller and Chad Brown - The TH0 language.
- Koen Claessen, Peter Baumgartner, and Stephan Schulz - The TF0 language.
- Jasmin Blanchette - Linking Sledgehammer to SystemOnTPTP, the TF1 language.
- Andrei Paskevich - The TF1 language.
- Cezary Kaliszyk and Florian Rabe - The TH1 language.
- Evgeny Kotelnikov - The TXF language.
- Christoph Benzüller and Alexander Steen - The NTF language.

Thank you!

⁹ See www.tptp.org/TPTP/TR/TPTPTR.shtml#Conclusion

11 Conclusion

This paper has described key components of the TPTP World that help make it a success, linking them together as “stepping stones” that lead from one component to another. The large number of citations to work of others, and explicitly Sect. 10, illustrate how the TPTP World has benefited, and benefited from, users in the ATP community. I am also grateful to the many people who have donated hard cash to the project, helping to keep it alive!

This paper has naturally focused on the successful parts of the TPTP World. There have also been some failed developments and suboptimal (in retrospect) decisions ☹. For example, in 2015 there was an attempt to develop a description logic form for the TPTP language. While some initial progress was made, it ground to a halt without support from the description logic community. A suboptimal design decision, rooted in the early days of the TPTP, is the naming scheme used for problem files. The naming scheme uses three digits to number the problems in each domain, thus setting a limit of 1000 problems, which failed to anticipate the numbers of problems that would be contributed to some of the problem domains. This has been overcome by creating multiple domain directories where necessary, but if it were to be done again, six or eight digit problem numbers shared across all domains would be an improvement.

The maintenance and development of the TPTP World is ongoing work. The most recent development is the languages and support for non-classical logics, initially modal logic [30,31]. The new format for representing interpretations (see Sect. 4) will be promulgated in the near future. As always, the ongoing success and utility of the TPTP problem library depends on ongoing contributions of problems – the automated reasoning community is encouraged to continue making contributions of all types of problems.

The TPTP World would not exist without the early strategic insights of Christian Suttner, with his willingness to let me do the organization without interference. Maybe his most wonderful contribution (which took him over two hours to produce when he was visiting me at James Cook University – I think he took a nap ☺) is his wonderfully simple plain-language definition of automated theorem proving: “the derivation of conclusions that follow inevitably from known facts”.

References

1. Bartocci, E., et al.: TOOLympics 2019: an overview of competitions in formal methods. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_1
2. Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 414–420. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_29

3. Blanchette, J., Urban, J. (eds.): Proceedings of the 3rd International International Workshop on Proof Exchange for Theorem Proving. No. 14 in EPiC Series in Computing, EasyChair Publications (2013)
4. Boyer, R., Lusk, E., McCune, W., Overbeek, R., Stickel, M., Wos, L.: Set theory in first-order logic: clauses for Godel's axioms. *J. Autom. Reason.* **2**(3), 287–327 (1986)
5. Cok, D., Stump, A., Weber, T.: The 2013 evaluation of SMT-COMP and SMT-LIB. *J. Autom. Reason.* **55**(1), 61–90 (2015)
6. Fuchs, M., Sutcliffe, G.: Homogeneous sets of ATP Problems. In: Haller, S., Simmons, G. (eds.) Proceedings of the 15th International FLAIRS Conference, pp. 57–61. AAAI Press (2002)
7. Fuenmayor, D., McKeown, J., Sutcliffe, G.: Towards StarExec in the cloud. In: Rawson, M., Schulz, S., Korovin, K. (eds.) Proceedings of the 15th International Workshop on the Implementation of Logics. p. To appear (2024)
8. Glickfield, B., Overbeek, R.: A foray into combinatory logic. *J. Autom. Reason.* **2**(4), 419–431 (1986)
9. Hoder, K., Voronkov, A.: Sine Qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 299–314. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_23
10. Hoos, H., Stützle, T.: SATLIB: an online resource for research on SAT. In: Gent, I., van Maaren, H., Walsh, T. (eds.) Proceedings of the 3rd Workshop on the Satisfiability Problem, pp. 283–292. IOS Press (2000)
11. Jech, T.: Otter experiments in a system of combinatory logic. *J. Autom. Reason.* **14**(3), 413–426 (1995)
12. Kaliszzyk, C., Sutcliffe, G., Rabe, F.: TH1: the TPTP typed higher-order form with Rank-1 polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning, pp. 41–55. No. 1635 in CEUR Workshop Proceedings (2016)
13. Laboratory, A.N.: The Argonne National Laboratory Problem Collection. <http://info.mcs.anl.gov/>
14. McCharen, J., Overbeek, R., Wos, L.: Problems and experiments for and with automated theorem-proving programs. *IEEE Trans. Comput.* **C-25**(8), 773–782 (1976)
15. McCune, W.: Otter 3.3 Reference Manual. Technical report. ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA (2003)
16. McCune, W., Padmanabhan, R.: Automated Deduction in Equational Logic and Cubic Curves. LNAI, vol. 1095. Springer-Verlag, Heidelberg (1996). <https://doi.org/10.1007/3-540-61398-6>
17. McCune, W., Wos, L.: Experiments in automated deduction with condensed detachment. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 209–223. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_167
18. Nieuwenhuis, R.: The impact of CASC in the development of automated deduction systems. *AI Commun.* **15**(2–3), 77–78 (2002)
19. Paulson, L., Blanchette, J.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) Proceedings of the 8th International Workshop on the Implementation of Logics, pp. 1–11. No. 2 in EPiC Series in Computing, EasyChair Publications (2010)
20. Peli, G., Bruggeman, J., Masuch, M., O Nuallain, B.: A Logical Approach to Formalizing Organizational Ecology: Formalizing the Inertia-Fragment in First-Order

- Logic. Technical report. CCSOM Preprint 92-74, Department of Statistics and Methodology, University of Amsterdam (1992)
21. Peli, G., Masuch, M.: The logic of propogation strategies: axiomatizing a fragment of organization ecology in first-order logic. In: Moore, D. (ed.) *Academy Of Management: Best Papers Proceedings 1994*, pp. 218–222 (1994)
 22. Pelletier, F.: Seventy-five problems for testing automatic theorem provers. *J. Autom. Reason.* **2**(2), 191–216 (1986)
 23. Peter, L., Hull, R.: *The Peter Principle*. Souvenir Press, Chicago (1969)
 24. Puzis, Y., Gao, Y., Sutcliffe, G.: Automated generation of interesting theorems. In: Sutcliffe, G., Goebel, R. (eds.) *Proceedings of the 19th International FLAIRS Conference*, pp. 49–54. AAAI Press (2006)
 25. Quaife, A.: Automated deduction in von Neumann-Bernays-Godel set theory. *J. Autom. Reason.* **8**(1), 91–147 (1992)
 26. Quaife, A.: *Automated Development of Fundamental Mathematical Theories*. Kluwer Academic Publishers (1992)
 27. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) *CADE 2019. LNCS (LNAI)*, vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
 28. Segre, A., Elkan, C.: A high-performance explanation-based learning algorithm. *Artif. Intell.* **69**(1–2), 1–50 (1994)
 29. SPRFN: The Problem Collection Distributed with the SPRFN ATP System. <https://www.cs.unc.edu/Research/mi/mi-provers.html>
 30. Steen, A., Fuenmayor, D., Gleißner, T., Sutcliffe, G., Benzmüller, C.: Automated reasoning in non-classical logics in the TPTP world. In: Konev, B., Schon, C., Steen, A. (eds.) *Proceedings of the 8th Workshop on Practical Aspects of Automated Reasoning*. p. Online. No. 3201 in *CEUR Workshop Proceedings* (2022)
 31. Steen, A., Sutcliffe, G.: TPTP world infrastructure for non-classical logics. In: Nalon, C., Steen, A., Suda, M. (eds.) *Proceedings of the 9th Workshop on Practical Aspects of Automated Reasoning*, p. Online. *CEUR Workshop Proceedings* (2024)
 32. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) *Proceedings of the 7th International Joint Conference on Automated Reasoning*. pp. 367–373. No. 8562 in *Lecture Notes in Artificial Intelligence* (2014)
 33. Sutcliffe, G.: SystemOnTPTP. In: McAllester, D. (ed.) *Proceedings of the 17th International Conference on Automated Deduction*, pp. 406–410. No. 1831 in *Lecture Notes in Artificial Intelligence*, Springer-Verlag (2000)
 34. Sutcliffe, G.: Semantic Derivation Verification: techniques and Implementation. *Int. J. Artif. Intell. Tools* **15**(6), 1053–1070 (2006)
 35. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) *CSR 2007. LNCS*, vol. 4649, pp. 6–22. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_4
 36. Sutcliffe, G.: The SZS ontologies for automated reasoning software. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, pp. 38–49. No. 418 in *CEUR Workshop Proceedings* (2008)
 37. Sutcliffe, G.: The TPTP problem library and associated infrastructure. The FOF and CNF parts, v3.5.0. *J. Autom. Reason.* **43**(4), 337–362 (2009)
 38. Sutcliffe, G.: The TPTP world – infrastructure for automated reasoning. In: Clarke, E.M., Voronkov, A. (eds.) *LPAR 2010. LNCS (LNAI)*, vol. 6355, pp. 1–12. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_1

39. Sutcliffe, G.: The TPTP process instruction language, with applications. In: Benzmüller, C., , Woltzenlogel Paleo, B. (eds.) Proceedings of the 11th Workshop on User Interfaces for Theorem Provers, pp. 1. No. 167 in Electronic Proceedings in Theoretical Computer Science (2014)
40. Sutcliffe, G.: The CADE ATP system competition - CASC. *AI Mag.* **37**(2), 99–101 (2016)
41. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* **59**(4), 483–502 (2017)
42. Sutcliffe, G.: The logic languages of the TPTP world. *Logic J. IGPL* **31**(6), 1153–1169 (2023)
43. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formaliz. Reason.* **3**(1), 1–27 (2010)
44. Sutcliffe, G., Kotelnikov, E.: TFX: the TPTP extended typed first-order form. In: Konev, B., Urban, J., Schulz, S. (eds.) Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning, pp. 72–87. No. 2162 in CEUR Workshop Proceedings (2018)
45. Sutcliffe, G., Kotthoff, L., Perrault, C., Khalid, Z.: An empirical assessment of progress in automated theorem proving. In: Benzmüller, C., Heule, M., Schmidt, R. (eds.) Proceedings of the 12th International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, p. To appear (2024)
46. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 406–419. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_32
47. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP language for writing derivations and finite interpretations. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 67–81. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_7
48. Sutcliffe, G., Seyfang, D.: Smart selective competition parallelism ATP. In: Kumar, A., Russell, I. (eds.) Proceedings of the 12th International FLAIRS Conference, pp. 341–345. AAAI Press (1999)
49. Sutcliffe, G., Steen, A., Fontaine, P.: The new TPTP format for interpretations. In: Korovin, K., Rawson, M., Schulz, S. (eds.) Proceedings of the 15th International Workshop on the Implementation of Logics, p. Submitted confidently (2024)
50. Sutcliffe, G., Suttner, C.: The TPTP problem library: CNF release v1.2.1. *J. Autom. Reason.* **21**(2), 177–203 (1998)
51. Sutcliffe, G., Suttner, C.: Evaluating general purpose automated theorem proving systems. *Artif. Intell.* **131**(1–2), 39–54 (2001)
52. Sutcliffe, G., Zimmer, J., Schulz, S.: Communication formalisms for automated theorem proving tools. In: Sorge, V., Colton, S., Fisher, M., Gow, J. (eds.) Proceedings of the Workshop on Agents and Automated Reasoning, pp. 52–57 (2003)
53. Trac, S., Puzis, Y., Sutcliffe, G.: An interactive derivation viewer. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of the 7th Workshop on User Interfaces for Theorem Provers. Electronic Notes in Theoretical Computer Science, vol. 174, pp. 109–123 (2007)
54. Van Gelder, A., Sutcliffe, G.: Extending the TPTP language to higher-order logic with automated parser generation. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 156–161. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_15

55. Weidenbach, C., et al.: System description: Spass version 1.0.0. In: CADE 1999. LNCS (LNAI), vol. 1632, pp. 378–382. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_34
56. Wilson, G., Minker, J.: Resolution, refinements, and search strategies: a comparative study. IEEE Trans. Comput. **C-25**(8), 782–801 (1976)

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


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



Theorem Proving and Tools



An Empirical Assessment of Progress in Automated Theorem Proving

Geoff Sutcliffe¹ , Christian Suttner², Lars Kotthoff³ ,
C. Raymond Perrault⁴ , and Zain Khalid¹ 

¹ University of Miami, Miami, USA
geoff@cs.miami.edu , zsk17@miami.edu

² Miami, USA

³ University of Wyoming, Laramie, USA
larsko@uwyo.edu

⁴ SRI International, Menlo Park, USA
ray.perrault@sri.com

Abstract. The TPTP World is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. This work uses data in the TPTP World to assess progress in ATP from 2015 to 2023.

Keywords: Automated Theorem Proving · Empirical Evaluation · Progress

1 Introduction

The TPTP World [69] (www.tptp.org) is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. The TPTP World includes the TPTP problem library, the TSTP solution library, standards for writing ATP problems and reporting ATP solutions, tools and services for processing ATP problems and solutions, and it supports the CADE ATP System Competition (CASC). This work uses data in the TPTP World to assess progress in ATP from 2015 to 2023.

Any meaningful assessment of progress in ATP must refer to the ability of ATP systems to solve problems. As the systems improve over time, the problems that they must solve also change to meet the demands of applications (with a fixed set of problems the systems can simply be finely tuned to the set, with inevitable asymptotic progress towards solving all the problems [52]). The TPTP problem library provides an evolving set of ATP problems that reflect the needs of ATP users, and is an appropriate basis for assessing the changing ability of ATP systems (the library is almost monotonically growing, but occasionally problems are removed – see Sect. 4.1). Alongside the TPTP problem library, the TSTP solution library provides data about ATP systems’ abilities to solve the problems in the TPTP problem library. This paper examines progress in ATP,

C. Suttner—Deceased.

© The Author(s) 2024

C. Benz Müller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 53–74, 2024.

https://doi.org/10.1007/978-3-031-63498-7_4

based on the data from TPTP v6.3.0 released on 28th November 2015 to TPTP v8.2.0 released on 13th June 2023. It is important to differentiate between evaluations at instances in time, such as provided by competitions, and evaluations over time. At instances of time the test problems used for evaluation, the systems being evaluated, and the hardware/software platform used, are static, e.g., as done in [20]. That provides a clean basis for a detailed comparison between systems. In contrast, evaluation over time is complicated by changing test problems, changing systems, and changing hardware/software. This dynamic evaluation environment requires additional control to provide meaningful results. The analyses done in this work do not explicitly factor in the resources needed to find solutions, e.g., hardware, time limits, etc.; Sect. 3.1 explains why this makes sense in the ATP context.

Related Work: The use of system performance data to evaluate a field of endeavour is common. In the realm of logic-based systems, examples include the various competitions [6] for logic-based systems (e.g., CASC [68], the SAT Competition [36], SMT-COMP [5], the ASP Competition [18]), longitudinal surveys of competitions [20, 75], the Technical Performance chapter of Stanford University’s AI Index Annual Report [45], the use of Shapley values to evaluate algorithmic improvements in SAT solving [25, 41], comparison of algorithmic and hardware advances (in SAT solving) [24], and other more specialized benchmarking, e.g., [89]. A general examination of the requirements for such benchmarking is provided by [9]. An ontology of artificial intelligence benchmarks is described in [14]. [52] provides an insightful analysis of the global dynamics of using benchmark sets in computer vision and natural language processing, and the takeaway messages are broadly applicable, including to benchmark sets for logic-based systems. In all cases the common measures for evaluation are (i) the ability of systems to solve problems, and (ii) the resources required by the systems to solve the problems. In order for results to be relevant, test problems must be representative of the problems the systems will face in applications, and the resource measurements must be appropriate for the availability and demands of the applications.

Summary of Findings: There has been progress in the last eight years, with stronger progress from v6.3.0 (2015) up to v7.1.0 (2018), but then a period of quiet until some more signs of progress in v8.2.0 (2023). There have been some first solutions of problems that are of direct interest to humans, a quite large number of first ATP solutions of problems from the TPTP, and some noteworthy improvements in individual ATP systems. There has been an apparent slowing of progress compared to the five years prior to 2015.

Paper Structure: Sections 2 and 3 provide a brief background to the TPTP problem library and TSTP solution library, highlighting features relevant to this work. Section 4 describes how the TPTP and TSTP data was prepared for analysis, and describes the measures used. Section 5 is the core of the paper, giving the results with commentary. Section 6 concludes.

Table 1. Overview of TPTP releases

Release	Date	Changes	Size	Analysed
v6.3.0	28/11/15	New TFO problems with arithmetic	20762	20168
v6.4.0	31/06/16	New problems	20897	20839
v7.0.0	24/07/17	First TH1 problems	21851	21310
v7.1.0	06/03/18	TXF syntax specified	22011	21893
v7.2.0	10/07/18	New problems	22026	21909
v7.3.0	02/08/19	New problems	22686	22570
v7.4.0	10/07/20	New problems	23291	23118
v7.5.0	13/07/21	New problems	24098	24027
v8.0.0	19/04/22	First TXF problems	24785	24027
v8.1.0	30/07/22	New problems	25257	25103
v8.2.0	13/06/23	New problems	25474	25325

2 The TPTP Problem Library

The core of the TPTP World is the TPTP problem library [66]; it is the de facto standard set of test problems for classical logic ATP systems. The problems can be browsed online¹ and documentation is available² Each release of the problem library is identified by a number in the form *version.edition.patch*. The current release at the time of writing was v8.2.0. Section 3 explains why the analyses of progress presented in this paper start at v6.3.0. Table 1 gives some summary data about the editions from v6.3.0 to v8.2.0. The Size column gives the number of problems in the release at the time of the release, while the Analysed column gives the number of problems left for analysis after the data cleaning described in Sect. 4.1. The acronyms for problem types are given in Sect. 2.1.

Each TPTP problem file has a header section (as comments) that contains information for users in four parts: the first part identifies and describes the problem; the second part provides information about occurrences of the problem in the literature and elsewhere; the third part provides semantic and syntactic characteristics of the problem; the last part contains comments and bugfix information. The third part is most relevant to this work. It contains the problem’s SZS status [77] that provides the semantic status of the problem, e.g., if it is a **Theorem**, a **Satisfiable** set of formulae, a problem whose status is **Unknown**, etc. It also includes statistics about the problem’s syntax, e.g., the number of formulae, the numbers of symbols, the use of equality and arithmetic, etc. The SZS status and the syntactic characteristics are used to form the Specialist Problem Class of the problem, as explained in Sect. 2.1.

¹ www.tptp.org/cgi-bin/SeeTPTP?Category=Problems.

² www.tptp.org/cgi-bin/SeeTPTP?Category=Documents.

2.1 Specialist Problem Classes

The problems in the TPTP library are divided into Specialist Problem Classes (SPCs) – classes of problems that are homogeneous wrt recognizable logical, language, and syntactic characteristics. Evaluation of ATP systems within SPCs makes it possible to say which systems work well for what types of problems. Empirically, homogeneity is ensured by examining the patterns of system performance across the problems in each SPC. For example, the separation of “essentially propositional” problems was motivated by observing that SPASS [85] performed differently on the ALC problems in the SYN domain of the TPTP. A data-driven test of homogeneity is also possible [26].

The characteristics used to define the SPCs in TPTP v8.2.0 are ...

1. TPTP language:
 - CNF – Clause Normal Form
 - FOF – First-Order Form
 - TFO – Typed Monomorphic First-order form
 - TF1 – Typed Polymorphic First-order form
 - TX0 – Typed Monomorphic eXtended First-order form
 - TX1 – Typed Polymorphic eXtended First-order form
 - TH0 – Typed Monomorphic Higher-order form
 - TH1 – Typed Polymorphic Higher-order form
2. SZS status:
 - THM – Theorem
 - CSA – CounterSATisfiabLe
 - CAX – Contradictory AXioms (merged with THM in this work)
 - UNS – UNSatisfiabLe
 - SAT – SATisfiabLe
 - UNK – UNKown
 - OPN – OPeN
3. Order (for CNF and FOF):
 - PRP – PRoPositional
 - EPR – Effectively PRoositional (known to be reducible to PRP)
 - RFO – Real First-Order (not known to be reducible to PRP)
4. Equality:
 - NEQ – No EQuality
 - EQU – EQuality (some or pure)
 - SEQ – Some (not pure) EQuality
 - PEQ – Pure EQuality
 - UEQ – Unit EQuality CNF
 - NUE – Non-Unit Equality CNF
5. Hornness (for CNF):
 - HRN – HoRN
 - NHN – Non-HoRN
6. Arithmetic (for T* languages):
 - NAR – No ARithmetic
 - ARI – ARithmetic.

Using these characteristics 223 SPCs are defined in TPTP v8.2.0. For example, the SPC TFO_THM_NEQ_ARI contains typed monomorphic first-order theorems that have no equality but include arithmetic. Combinations of SPCs are written using UNIX globbing, e.g., TFO_THM_*_NAR is the combination of TFO_THM_EQU_NAR and TFO_THM_NEQ_NAR – typed monomorphic higher-order theorems problems, either with or without equality, but no arithmetic.

The SPCs are used when computing the TPTP problems difficulty ratings, as explained in Sect. 2.2.

2.2 TPTP Problem Ratings

Each TPTP problem has a difficulty rating that provides a well-defined measure of how difficult the problem is for current ATP systems [76]. The ratings are based on performance data in the TSTP (see Sect. 3), and are updated in each TPTP edition. Rating is done separately for each SPC. First, a partial order between systems is determined according to whether or not a system solves a strict superset of the problems solved by another system. If a strict superset is solved, the first system is said to *subsume* the second. Then the fraction of non-subsumed systems that fail on a problem is the difficulty rating for the problem. Problems that are solved by all of the non-subsumed systems get a rating of 0.00 (“easy”); problems that are solved by some of the non-subsumed systems get a rating between 0.00 and 1.00 (“difficult”); problems that are solved by none of the non-subsumed systems get a rating of 1.00 (“unsolved”).

3 The TSTP Solution Library

The complement of the problem library is the TSTP solution library [65,67]. The TSTP is built by running all the ATP systems that are available in the TPTP World on all the problems in the TPTP problem library. At the time of writing this paper, the TSTP contained the results of running 87 ATP systems and system variants on all the problems in the TPTP that they could attempt. This produced 1091026 runs, of which 432718 (39.6%) solved the problem. One use of the TSTP is for ATP system developers to examine solutions to problems and thus understand how they can be solved, leading to improvements to their own systems. The use considered here is for TPTP problem ratings.

Prior to 2010 the data in the TSTP came from results submitted by ATP system developers, who performed testing on their own hardware. From 2010 to 2013 the data was generated on the TPTP World servers at the University of Miami. Since 2014 the ATP systems have been run on StarExec [63], initially on the StarExec Iowa cluster, and since 2018 on the StarExec Miami cluster. StarExec has provided stable platforms that produce reliably consistent and comparable data in the TSTP. The analyses presented in Sect. 4 start at TPTP v6.3.0, which was released in November 2015. By that time the problem ratings were based on data produced on the StarExec computers.

The StarExec Iowa computers have a quad-core Intel Xeon CPU E5-2609 CPU running at 2.40 GHz, 128 GiB memory, and the CentOS Linux release 7.9.2009 operating system. The StarExec Miami computers have an octa-core Intel Xeon E5-2667 v4 CPU running at 2.10 GHz, 128 GiB memory, and the CentOS Linux release 7.4.1708 operating system. One ATP system is run on one CPU at a time, with a 300 s CPU time limit and a 128GiB memory limit (see Sect. 3.1). The minor differences between the Iowa and Miami configurations can be ignored for the task of “solving problems”, as is explained in Sect. 3.1.

3.1 Resource Limits

Analysis shows that increasing resource limits does not significantly affect which problems are solved by an ATP system. Fig. 1 illustrates this point; it plots the CPU times taken by several contemporary ATP systems to solve the TPTP problems for the FOF_THM_RFO_* SPCs, in increasing order of time taken. The relevant feature of these plots is that each system has a point at which the time taken to find solutions starts to increase dramatically. This point is called the system’s Peter Principle [55] Point (PPP), as it is the point at which the system has reached its level of incompetence. Evidently a linear increase in the computational resources beyond the PPP would not lead to the solution of significantly more problems. The PPP thus defines a realistic computational resource limit for the system. Therefore, provided that enough CPU time and memory are allowed for an ATP system to reach its PPP, a usefully accurate measure of what problems it can solve is achieved. The performance data in the TSTP is produced with adequate resource limits.

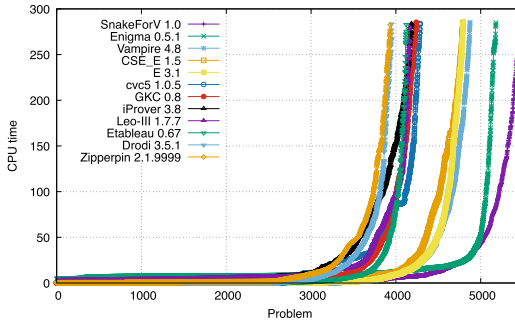


Fig. 1. CPU times for FOF_THM_RFO_*

4 Analysis Processes

4.1 Analysis Data

The analyses performed in this assessment use the TPTP problem ratings, and historical data about which ATP systems solved which problems in each TPTP release. The data was extracted from the `ProblemAndSolutionStatistics` file that accompanies each TPTP release, which summarizes information from the header fields of the TPTP problem files and corresponding TSTP solution files. As explained in Sect. 3, TSTP data starting from TPTP v6.3.0 in November 2015 has been used, taking snapshots at each TPTP edition up to v8.2.0.

Before analysis the rating data was cleaned as follows:

Cleaning for Bias: The TPTP tags problems that are designed specifically to be suited or ill-suited to some ATP system, calculus, or control strategy as *biased*. The biased problems were excluded from the analyses.

Cleaning for Bugfixes: Over time some problems have had to be removed from the TPTP because they are renamed, duplicates, wrongly formulated, etc. Such problems in a TPTP release are thus not in subsequent releases. The removed problems were excluded from the analyses.

Cleaning for the Past: Problems are added to the TPTP in each release, and corresponding TSTP data is generated using the available ATP systems. As it is not possible to run all previously available ATP systems on new problems when they are added to the TPTP, it has been (quite reasonably) assumed that if a problem was unsolved by the current ATP systems when it was added to the TPTP (initial rating 1.00), then it would have been unsolved by previously available ATP systems. The rating data was thus augmented for problems that were added after v6.3.0 and had an initial rating of 1.00, by setting the problems' ratings in the prior TPTP releases to 1.00. There were 1854 such problems. This, however, can lead to an unfairly optimistic view of progress, because those retrospectively added 1.00 ratings increase the average problem rating in the past. For problems that were solved when they were added to the TPTP (initial rating less than 1.00), it is unknown if the previously available ATP systems would have been able to solve them. Augmenting the rating data by setting the problems' ratings in prior TPTP releases to their initial rating of less than 1.00 could lead to an optimistic or pessimistic view of progress, depending if the rating was greater or less than the average in the past releases. In this work the rating data was augmented for problems that were added after v6.3.0 and had an initial rating less than 1.00, by setting the problems' ratings in the prior TPTP releases to their initial rating. There were 2632 such problems. The optimistic/pessimistic effect gets stronger when rating data is augmented for problems that were added in more recent TPTP releases. A total of $1854 + 2632 = 4486$ problems had their initial ratings propagated backwards, starting from the various releases over the eight years of analysis. Overall this could have had a slightly optimistic impact in the analyses.

Cleaning for Change: A counterintuitive feature of an individual problem's difficulty ratings is that they sometimes increase with time. It is counterintuitive because the problem has not changed. (This was also noted in a prior analysis [69].) Increases are caused by new ATP systems or system versions becoming available. If a new system is not subsumed then its TSTP data is used in the rating process: the ratings of problems that it solves decrease, but at the same time the ratings of problems that it does not solve increase – you have to “pay the

piper”.³ A common instance of this phenomenon is a new system that can solve some previously unsolved (rating 1.00) problems, but that cannot solve a substantial number of problems that are solved by other systems (rating less than 1.00). In this work the anomaly is resolved by additionally looking at *monotonic ratings*: if a problem’s rating in a TPTP release is greater than its previous rating, the monotonic rating is set to the previous lower rating. Monotonized ratings make clear sense in the case of problems that were unsolved (rating 1.00) and were later solved by a new system (the rating drops to less than 1.00) – if a problem is solved, it cannot become unsolved – the solving system still exists in principle. In cases where the rating is less than 1.00 monotonized ratings might be considered to be optimistic because ratings do have to “pay the piper”.

4.2 Coherent SPC Sets

Five of the analyses performed (see Sect. 4.3) require data from sets of problems with similar characteristics, so that the analysis results are wrt that type of problem. The basis for such sets is the SPCs (see Sect. 2.1), which provide a fine-grained partitioning of the TPTP problems so that each SPC is coherent. Some SPCs that capture compatible problem characteristics can be merged to form a *coherent SPC set*.

The coherent SPC sets used for the analyses are listed in Table 2. The SPC set column lists the SPCs that are in the set, using the abbreviations given in Sect. 2.1. Some noteworthy exclusions are: typed extended first-order problems, because they were added to the TPTP only in v8.0.0; typed polymorphic first-order and higher-order problems, because too few systems are capable of attempting the problems and generating the necessary TSTP data; some SPCs that have too few problems, e.g., TFO_CSA*_NAR and TFO_SAT*_NAR, which combined have only 154 problems.

4.3 Six Analyses

The cleaned TPTP problems ratings and historical TSTP data has been used for six analyses of progress in ATP. Individual problem ratings are used for the first analysis. The other five analyses are wrt the coherent SPC sets described in Sect. 4.2.

First Solutions: Arguably the most successful use of ATP comes from the “hammers” [15] associated with Interactive Theorem Proving (ITP) systems, where the individual problems being solved are typically not of direct interest to the human users who are focussed on the larger task being addressed in the ITP system. In contrast, the use of ATP by practitioners to solve individual problems

³ Conversely, if a system that was not subsumed becomes unavailable, it no longer contributes TSTP data for new problems. This phenomenon is rare (e.g., Isabelle ran fine on StarExec Iowa but did not port to StarExec Miami in 2018) and has not materially impacted the analyses of progress.

Table 2. Coherent SPC sets

SPC set	Description
CNF_UNR_RFO_PEQ_UEQ	Unsatisfiable really-first-order unit equality clauses
CNF_UNR_RFO_NEQ_* \cup CNF_UNR_RFO_SEQ_* \cup CNF_UNR_RFO_PEQ_NUE	Unsatisfiable really first-order clauses that are not unit equality
CNF_SAT_RFO_*	Satisfiable really-first-order clauses
FOF_*_PRP \cup FOF_*_EPR_* \cup CNF_*_PRP \cup CNF_*_EPR_*	Unsatisfiable and satisfiable propositional and effectively propositional clauses. Un/Satisfiable is coherent because the problems are decidable
FOF_THM_RFO_*	Really first-order theorems, with or without equality
FOF_CSA_RFO_* \cup FOF_SAT_RFO_*	Really first-order non-theorems and satisfiable sets, with and without equality
TFO_THM_*_NAR	Typed monomorphic first-order theorems, with and without equality, no arithmetic
TFO_THM_*_ARI	Typed monomorphic first-order theorems, with and without equality, with arithmetic
THO_THM_*_NAR	Typed monomorphic higher-order theorems, with and without equality, no arithmetic

that have resisted manual approaches is less common and possibly less successful, but the sparsity makes successes particularly noteworthy. First solutions of problems that are of direct interest to humans are indications of progress. Such problems are identifiable by (i) the rating decreasing from 1.00, and (ii) evidence that the problem is of direct interest to some humans.

Average Difficulty Ratings: This is the average problem difficulty rating, and the average monotonized difficulty rating. (This approach was used in [73].) As the problems are unchanged (they are not actually getting easier), decreases are evidence of progress in ATP systems.

Never-Solved: This is the fraction of problems that were unsolved (rating 1.00) in all TPTP releases up to each TPTP release, relative to the number in v6.3.0. (The converse of this is plotted in [78].) Decreases are evidence of progress.

Solved: For the given system and a given TPTP release, this is ...

$$\frac{\textit{ProblemSolvedInRelease} - \textit{LeastSolvedAcrossAllReleases}}{\textit{MostSolvedAcrossAllReleases} - \textit{LeastSolvedAcrossAllReleases}}$$

The releases with a 1.00 value are those in which the most problems were solved, and those with 0.00 had the least number solved. Increases are evidence of progress.

Always-Easy: This is the converse of Never-solved – the fraction of problems that were easy (rating 0.00) in all TPTP releases back to each TPTP release, relative to the number in v8.2.0. Increases are evidence of progress.

Shapley Value: A State-of-the-Art (SotA) ATP system for a TPTP release is defined as one that solves the union of the problems solved by the individual ATP systems, e.g., by using competition parallelism [79]. The Shapley value [87] is the average of the marginal contributions (how much the SotA system improves when adding each given system) over all systems added to all possible subsets of other systems. First, temporal Shapley analysis [41] is used to measure the SotA systems' contributions to progress, normalized by the number of previously unsolved problems so that 0.0 means no previously unsolved problems were solved and 1.0 means all previously unsolved problems were solved. Peaks indicate stronger progress. Next, (non-temporal) Shapley analysis [25] is used to measure the contributions of the individual systems in each release. Finally, temporal Shapley analysis for all systems in all releases is used to measure the contributions of the individual system versions when they were introduced. The latter two analyses were used to provide insights for the commentary about the systems' performances (they are not plotted in Sect. 5).

5 Evidence of Progress

5.1 First Solutions

There are some nice examples of ATP systems finding first solutions to problems that are of direct interest to humans ...

- Model finding ATP systems were used to solve previously open problems concerning the existence of quasigroups satisfying certain additional conditions [61]. Many examples are in the GRP domain of the TPTP.
- The solution of the Robbins problem⁴ by the specialist ATP system EQP [47] in 1996 was a noteworthy success, as the problem had defied the efforts of eminent mathematicians [29]. It is ROB001-1 in the TPTP, and still has a rating of 1.00 because it has not been solved by a non-specialist ATP system.

⁴ The Robbins problem was posed in personal communications between Edward Huntington, Herbert Robbins, and Alfred Tarski. The background is given in en.wikipedia.org/wiki/Robbins_algebra.

- The first inner five-segment theorem of Tarski’s geometry [60] was first automatically proved by E [58] in 2019, after being posed by Quaife in 1989 [56]. It is problem GE0033-2 in the TPTP.
- The proof of the consistency of an encoding of a large fragment of a high school textbook on biology [19] by iProver [38] in 2021 showed how new techniques could be used to find models in large theories. It is problem BI0001+1 in the TPTP.
- Larry Wos’ challenge to find a “circle of pure proofs” that shows the equivalence of the four Moufang identities [86] was met by careful application [81] of Otter [48] in 2021. While those specific problems are not in the TPTP, many related problems are in the ring theory (RNG) domain of the TPTP.

5.2 Solutions and Ratings

A total of 25325 problems were analysed over the coherent SPCs, of which 19762 (78%) were solved in TPTP v6.3.0, increasing to 20227 (80%) in v8.2.0. Of the 25325 problems, 5563 (22%) were unsolved when they were added to the TPTP, of which 1009 (4%) were solved in some release by v8.2.0. Conversely, there were 8984 problems (35%) that had a rating of 0.00 in v8.2.0, of which 2965 (12%) had a higher rating in some preceding release. These overall figures provide evidence of overall progress, but the contributions vary across the coherent SPC sets. Figures 2, 3, 4, 5, 6, 7, 8, 9 and 10 plot the values for each coherent SPC set for the latter five analyses described in Sect. 4.3.⁵ The captions provide the numbers of ‘P’roblems in TPTP v8.2.0, the number left for analysis after the data cleaning, and the numbers of ‘N’ever-solved, ‘S’olved, and ‘A’lways-easy problems in releases v6.3.0-v8.2.0.

Figures 2, 3 4, 5, 6 and 7 plot the values for the CNF- and FOF-based coherent SPC sets. CNF is now the “assembly language” of most ATP systems, which typically translate more expressive logics down to CNF. As such, progress in CNF typically contributes to progress in other SPCs.

CNF_UNRS_RFO_PEQ_UEQ showed progress in v6.4.0 due to the strong performance of Twee 2.0 [62], which made a lot more problems always-easy by v7.0.0. Also in v6.4.0, Waldmeister 710 [44] solved five problems that had never been solved before. In v7.4.0 E 2.5 made a strong contribution, then in v8.1.0 Twee 2.4 made another strong contribution, alongside CSE_E 1.3 [88]. Waldmeister 710 had the highest Shapley value across all the releases, but in v8.1.0 both Twee and CSE_E solved more problems than Waldmeister. The lowest number of problems solved was in v7.5.0 and v8.0.0, when 23 fewer problems were solved than in v7.4.0 – not many in the context of the 1034 solved in v7.4.0. The only discernible common feature of those 23 problems is that they had ratings over 0.90 in v7.4.0. Apparently some changes in the ATP system versions from v7.4.0 to v7.5.0 made the problems unsolvable in v7.5.0, and further changes reversed the situation for v8.1.0 when 1043 problems were solved.

⁵ Data: github.com/GeoffsPapers/ATPPProgress2024/raw/master/DataForAnalysis.

CNF_UNRS_RFO_*_NUE had a small but quite consistent decline in the problem ratings, indicating some progress. The big advances were in v7.0.0 when Vampire 4.2 performed well, including solving 33 problems that had never been solved before. In v8.2.0 SnakeForV 1.0 solved 26 problems that had never been solved before. The biggest drop in problems solved was between v7.2.0 and v7.3.0, when 66 fewer problems were solved. The largest increase in problems solved was between v8.1.0 and v8.2.0, when 50 more problems were solved. SnakeForV was again the big contributor to the increase. SnakeForV is interesting, as it is a variant of Vampire with an independent reimplementa-tion of Spider-style [82] strategy discovery and schedule construction that factors in prover randomiza-tion [64].

CNF_SAT_RFO_* had only one high point, in v6.4.0 when Vampire 4.0.5 made a strong contribution, including solving four problems that had never been solved before. The sudden drop in problems solved in v7.0.0 was due to Prover9 1105 [46] data not being available; the reason is lost in the mists of time, but it is interesting to note that the older system was able to solve some problems that other systems could not. By v7.1.0 new systems had taken up the slack. The plots are all quite stable from v7.1.0 onwards.

{FOF,CNF}_*_EPR_* had two points of progress, the first in v7.0.0 and the second in v7.3.0. In v7.0.0 the progress came from iProver 2.6 that had integrated an abstraction-refinement framework [30], and Vampire 4.2 that had some changes in its model building. Between them they solved five problems that had never been solved before. In v7.3.0 iProver 3.0 integrated superposition [23]. The number of problems solved increased continuously until v8.2.0. The drop in v8.2.0 was due to poorer performances by the new iProver 3.7, SnakeForV 1.0, and Vampire 4.7. These systems share the same FOF to CNF translator, which might have been the source of the common change.

FOF_THM_RFO_* is the best known of the FOF-based SPCs, with the most ATP systems able to attempt the problems, and is the target of most new systems. The problem difficulty ratings are quite flat, but the number of problems solved increased quite regularly, from 6086 in v6.3.0 to 6235 in v8.2.0. The largest step of progress came in v7.0.0 when Vampire 4.2 solved 72 problems that had never been solved before, thanks to improvements in preprocessing. ET 0.2 [37] also contributed to the progress in v7.0.0. In v7.4.0 Enigma 0.4 [32,33] was a new system that made a strong contribution to progress. Vampire 4.5 also contributed to progress in v7.4.0, with a new layered clause selection approach [27] and a new subsumption demodulation rule [28].

FOF_{CSA,SAT}_RFO_* is also well known, and along with its typed first-order counterpart (not analysed due to insufficient data) is important for applications, e.g., [22]. The largest sign of progress was in v6.4.0. The main contributors were Vampire 4.0.5 with improvements to its satisfiability checking, and iProver 2.5 with restructured core data structures and improved preprocessing including predicated elimination. Vampire 4.0.5 solved 10 problems that had never been solved before. There is a drop of 10 problems solved from v8.0.0 to v8.2.0. As in CNF_UNRS_RFO_PEQ_UEQ, there is no discernible common feature of those 10

problems, and their ratings were at most 0.75. This again shows that the set of problems solved by evolving versions of systems does not grow monotonically.

Figures 8, 9 and 10 plot the values for the TFF- and THF-based coherent SPC sets. TFO_THM_*_NAR uses the simplest of the typed TPTP languages. In v7.0.0 there was progress thanks to Vampire 4.2 and CVC4 1.5.2 [4]. In v8.2.0 there was progress thanks to SnakeForV 1.0. In between those points of progress there was a drop in the number of problems solved, from 282 in v7.4.0 down to 260 in v7.5.0, apparently due to poorer performance of CVC4 1.9 in v7.5.0 compared to that of CVC4 1.7 in v7.4.0.

TFO_THM_*_ARI is important because it uses the simplest TPTP language that includes arithmetic, which occurs naturally in application areas [16, 39, 53]. There was clearly some significant progress in v6.4.0 as many problems were solved for the first time by Vampire 4.0.5, which had integrated Z3 [50] since Vampire 4.0. This contributed to the increase in the number of problems solved, from 915 in v6.3.0 to 1009 in v6.4.0. CVC4 1.5 [4] and Princess 150706 [57] also performed well.

THO_THM_*_NAR uses typed higher-order logic, and despite using a more expressive language than the TFO_* SPCs, has been the focus of ATP system development longer [70, 74]. The problem ratings declined moderately, and there were bursts of progress in v7.0.0 and v7.5.0. The progress in v7.0.0 was largely thanks to Satallax 3.2 [17], which included a SInE-like [31] procedure for premise selection that enabled it to solve some large problems that were previously out of reach. That progress increased the number of always-easy problems by v7.1.0. In v7.5.0 Zipperposition 2.0 [8] improved over the previous version, and solved 18 problems that had never been solved before.

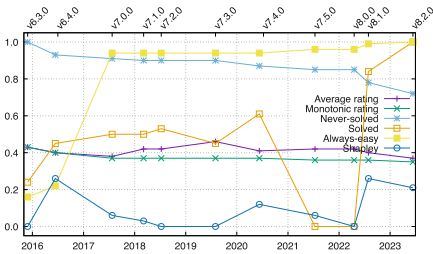


Fig. 2. CNF_UNRS_RFO_PEQ_UEQ P:1140-1140 N:120-86 S:1020-1049 A:38-233

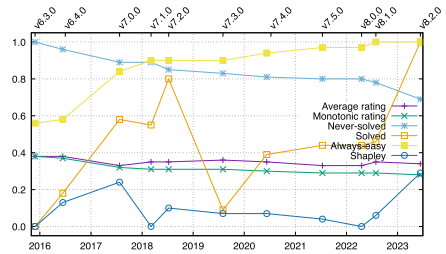


Fig. 3. CNF_UNRS_RFO_*_NUE P:4445-4441 N:569-391 S:3873-3966 A:1004-1780

Figure 11 was presented (verbatim) in a prior analysis done at TPTP release v6.4.0 [69]. The figure plotted the average ratings for the 14527 problems that were unchanged in the TPTP since v5.0.0, and whose ratings had not been stuck at 0.00 or 1.00 since v5.0.0. It was noted in [69]: “The ratings generally show a downward trend - there has been progress!”. Figure 12 shows the same done at TPTP release v8.2.0, for the 16236 problems that were unchanged in the TPTP since v6.3.0, and whose ratings have not been stuck at 0.00 or 1.00 since v6.3.0.

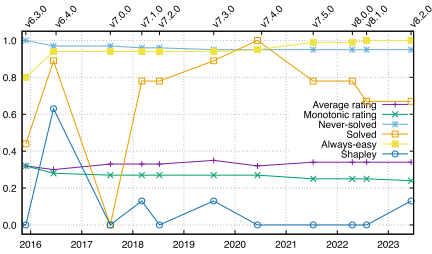


Fig. 4. CNF_SAT_RFO_* P:1044-1042 N:155-147 S:887-889 A:476-598

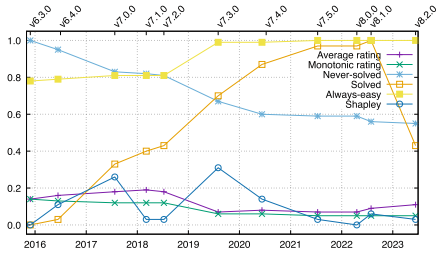


Fig. 5. {FOF,CNF}_*_EPR_* P:1457-1425 N:78-43 S:1347-1360 A:1027-1311

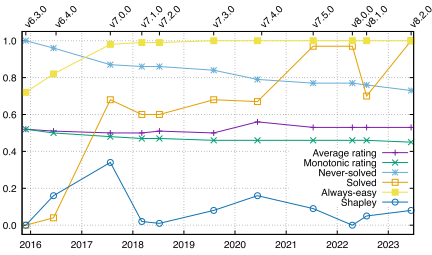


Fig. 6. FOF_THM_RFO_* P:7204-7202 N:1116-818 S:6086-6235 A:696-971

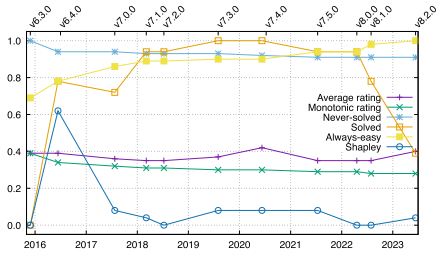


Fig. 7. FOF_{CSA,SAT}_RFO_* P:1329-1028 N:282-256 S:746-753 A:481-709

The two figures' plots dovetail quite well, which gives confidence that they really are comparable (there are some minor differences caused by the data cleaning done for this work, and recent refinements to the rating calculations [71, 72]). The older plots show a quite clear downward trend both overall and for the four types of problems, while the new plots do not. Possible reasons are discussed in the conclusion (Sect. 6).

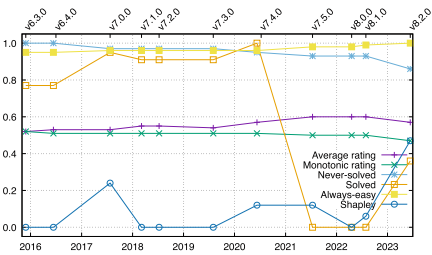


Fig. 8. TFO_THM_*_NAR P:400-397 N:120-103 S:277-268 A:117-123

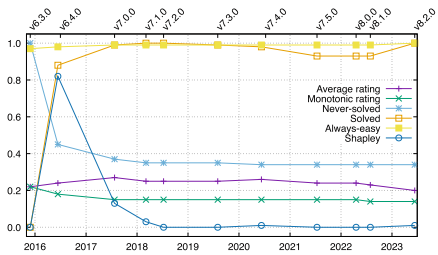


Fig. 9. TFO_THM_*_ARI P:1176-1087 N:172-58 S:915-1022 A:763-785

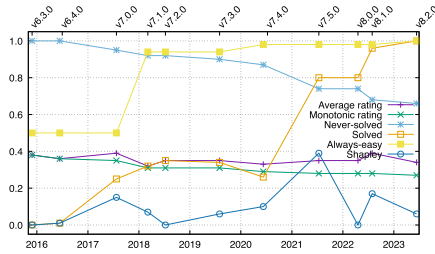


Fig. 10. THO_THM*_NAR PA:3189-3183 N:461-305 S:2722-2814 A:617-1244

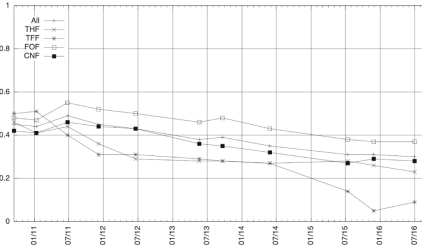


Fig. 11. Ratings from v5.0.0 to v6.4.0

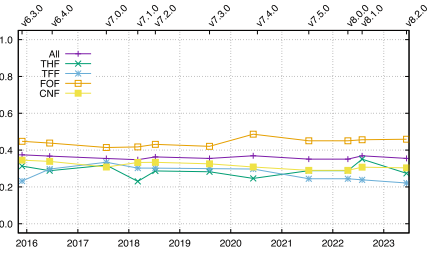


Fig. 12. Ratings from v6.3.0 to v8.2.0

6 Conclusion

This paper has presented an empirical assessment of progress in ATP, using data from the TPTP World in TPTP v6.3.0 in 2015 to v8.2.0 in 2023. The assessment has been in terms of six measures, divided into nine coherent SPC sets of problems that are reasonably homogeneous for ATP systems. The assessment shows that there has been progress in the last eight years, with stronger progress from v6.3.0 (2015) up to v7.1.0 (2018), but then a period of quiet until some more signs of progress in v8.2.0 (2023). There have been some first solutions of problems that are of direct interest to humans, and a quite large number of first ATP solutions of problems from the TPTP. The coherent SPCs with the strongest signs of progress were CNF_UNRS_RFO_PEQ_UEQ and THO_THM*_NAR.

In addition to overall trends, it is worth noting some of the salient improvements in individual ATP systems, extracted from Sect. 5 . . .

- The development of EQP leading to the solution of the Robbins problem in 1996.
- The release of Waldmeister in 1997 (before the period of analysis), which dominated UEQ problem solving until the arrival of Twee 2.4 in 2021.
- The release of Satallax 2.8 in 2015 with strong performance on THF problems, improving up to Satallax 3.2 in 2017.
- The release of Vampire 4.0.5 in 2016, with arithmetic capability included.
- The release of Vampire 4.2 in 2017 with significantly improved performance on many types of problems, including NUE, EPR, FOF, and TF0_NAR.

- The release of iProver versions 2.5 to 2.8 between 2016 and 2018, with strong performance on EPR problems.
- the release of Zipperposition 2.0 in 2020, with strong performance on THF problems.
- The release of the Vampire-based SnakeForV 1.0 in 2022, which outperformed Vampire on many types of problems.

In terms of problem difficulty ratings, the monotonized ratings necessarily went down but the trend was not dramatic, and the raw ratings were generally stable. This is in contrast to the clearly decreasing ratings from 2011 to 2016. The reasons for that apparent slowing of progress are not definitely known, but we have thought of the following possible reasons:

- System developers have expended effort adding breadth of capability at the expense of depth, e.g., E – processed only CNF and FOF up to 2015, added TF0 in 2017 [59], TX0 and TH0 in 2019 [83]; iProver – processed only CNF and FOF up to 2015, added TF0 with arithmetic in 2021 (unpublished); Vampire – processed only CNF, FOF, and TF0 up to 2015, added TX0 in 2016 [40], TF1 and TX1 in 2020 [13], and THF in several incarnations from 2019 to 2023 [10–12].
- The entry barrier to building new high-performance ATP systems is high, because top systems dominate the field and attract the best developer talent. In Maria Paola Bonacina’s welcoming address at the Dagstuhl Seminar “The Next Generation of Deduction Systems: from Composition to Compositionality”⁶ she referred to this as a “crisis of growth”.
- New systems that take new approaches that solve different subsets of SPCs have an impact on problem difficulty ratings. For examples: CSE_E [88] was new in 2018, combining the S-CS calculus with E; Zipperposition [7] was new in 2019, extending superposition to higher-order logic; Twee [62] was new in 2018, solving CNF and FOF problems by transformation to UEQ.
- Time spent on machine learning based techniques, for axiom selection, e.g., [42, 80], given clause selection, e.g., [1, 21, 34, 49]), learning for large problem corpora, e.g., [3, 35, 43], and use of large language models to improve ATP performance [2, 84], is focussed largely on sets of many quite similar problems over one fixed signature. The progress made in that usage does not contribute directly to general progress in solving individual problems with different signatures, as measured in this work.
- SMT solvers have been in existence since the late 1970s [51], blossomed fully in the early 2000s, and has attracted ever-increasing interest since then. Some ATP systems have been adapted to solving SMT problems, e.g., Vampire has been entered into SMT-COMP since 2016, and iProver since 2021. This is all good work, but has possibly diverted developer energy from ATP to SMT.
- The divisions of CASC cause developers to put extra effort into solving the types of problems in the division. For examples, the Effectively Propositional (EPR) division was run from CASC-JC in 2001 to CASC-27 in 2019, and

⁶ www.dagstuhl.de/23471.

during those years several ATP systems were optimized for EPR problems, most notably iProver. Putting a division on hiatus leads to less development in that aspect of ATP.

- In [72] it was noted that CASC might be causing incremental development of ATP systems. This concern has been expressed as far back as CASC-JC in 2001 [54]. In response to this concern CASC-J12 will have a new ICU (I Challenge yoU) division that focusses on solving hard problems rather than solving more problems, hoping to stimulate new developments and progress.

This assessment of progress is based on ATP systems' abilities to solve problems. Evaluation of other performance measures would be interesting, e.g., stability of proof search modulo perturbations of the input, and some have been done in other evaluations of logic-based systems. These include measures such as resource usage and verifiability of proofs/models. Evaluation of non-performance measures is often ignored, but for users might be just as necessary. These include measures such as the range of logics covered, ease of building and deploying, portability to different hardware and operating system environments, availability of source code, quality of source code and its documentation, licensing that permits a required level of use or modification, availability of user documentation, and (maybe most importantly!) developer support. These are topics for future assessments.

References

1. Aygün, E., et al.: Proving theorems using incremental learning and hindsight experience replay. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine Learning, pp. 1198–1210. No. 162 in Proceedings of Machine Learning Research (2022)
2. Azerbayev, Z., et al.: Llemma: An Open Language Model For Mathematics (2023). [arXiv:2310.10631](https://arxiv.org/abs/2310.10631)
3. Bansal, K., Loos, S., Szegedy, C., Wilcox, S.: HOList: an environment for machine learning of higher-order theorem proving. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, pp. 454–463 (2019)
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: satisfiability modulo theories competition. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 20–23. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_4
6. Bartocci, E., et al.: TOOLympics 2019: an overview of competitions in formal methods. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_1
7. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 396–412. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_23

8. Bentkamp, A., Blanchette, J., Tournet, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 55–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_4
9. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transfer* **21**, 1–29 (2019)
10. Bhayat, A.: Automated theorem proving in higher-order logic. Ph.D. thesis, Faculty of Science and Engineering, University of Manchester, Manchester, United Kingdom (2020)
11. Bhayat, A., Rawson, M., Schoisswohl, J.: Superposition with delayed unification. In: Pientka, B., Tinelli, C. (eds.) CADE 2023. LNCS, vol. 14132, pp. 23–40. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_2
12. Bhayat, A., Reger, G.: Restricted combinatory unification. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 74–93. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_5
13. Bhayat, A., Reger, G.: A polymorphic vampire. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 361–368. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_21
14. Blagec, K., Barbosa-Silva, A., Ott, S., Samwald, M.: A curated, ontology-based, large-scale knowledge graph of artificial intelligence tasks and benchmarks. *Sci. Data* **9**(322), 1–10 (2022)
15. Blanchette, J., Kaliszzyk, C., Paulson, L., Urban, J.: Hammering towards QED. *J. Formaliz. Reason.* **9**(1), 101–148 (2016)
16. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let’s verify this with why3. *Int. J. Softw. Tools Technol. Transfer* **17**(6), 709–727 (2015)
17. Brown, C.E.: Satallax: an automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 111–117. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_11
18. Calimeri, F., Ianni, G., Krennwallner, T., Ricca, F.: The answer set programming competition. *AI Mag.* **33**(4), 114 (2012)
19. Chaudri, V., Dinesh, N., Inclezan, D.: Three lessons in creating a knowledge base to enable explanation, reasoning and dialog. In: Klenk, M., Laird, J. (eds.) Proceedings of the 2nd Annual Conference on Advances in Cognitive Systems, pp. 187–203 (2013)
20. Cok, D., Stump, A., Weber, T.: The 2013 evaluation of SMT-COMP and SMT-LIB. *J. Autom. Reason.* **55**(1), 61–90 (2015)
21. Crouse, M., et al.: A deep reinforcement learning approach to first-order logic theorem proving. In: Leyton-Brown, K., Mausam (eds.) Proceedings of the 35th AAAI Conference on Artificial Intelligence, vol. 35, no. 7, pp. 6279–6287. AAAI Press (2021)
22. D’Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **27**(7), 1165–1178 (2008)
23. Duarte, A., Korovin, K.: Implementing superposition in iprover (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 388–397. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_24
24. Fichte, J.K., Hecher, M., Szeider, S.: A time leap challenge for SAT-solving. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 267–285. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_16

25. Fréchet, A., Kotthoff, L., Michalak, T., Rahwan, T., Hoos, H., Leyton-Brown, K.: Using the shapley value to analyze algorithm portfolios. In: Schuurmans, D., Wellman, M. (eds.) Proceedings of the 30th AAAI Conference on Artificial Intelligence, pp. 3397–3403. AAAI Press (2016)
26. Fuchs, M., Sutcliffe, G.: Homogeneous sets of ATP problems. In: Haller, S., Simmons, G. (eds.) Proceedings of the 15th International FLAIRS Conference, pp. 57–61. AAAI Press (2002)
27. Gleiss, B., Suda, M.: Layered clause selection for theory reasoning. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 402–409. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_23
28. Gleiss, B., Kovács, L., Rath, J.: Subsumption demodulation in first-order theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 297–315. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_17
29. Henkin, L., Monk, J., Tarski, A.: Cylindrical Algebras, vol. Part 1. North-Holland (1971)
30. Hernandez, J., Korovin, K.: Towards an abstraction-refinement framework for reasoning with large theories. In: Eiter, T., Sands, D., Schulz, S., Urban, J., Sutcliffe, G., Voronkov, A. (eds.) Proceedings of the IWIL Workshop and LPAR Short Presentations. No. 1 in Kalpa Publications in Computing (2017)
31. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 299–314. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_23
32. Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., Urban, J.: ENIGMA anonymous: symbol-independent inference guiding machine (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 448–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_29
33. Jakubův, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of Certified Programs and Proofs 2017, pp. 43–52. ACM (2017)
34. Jakubův, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 292–302. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_20
35. Jakubův, J., Urban, J.: Hammering mizar by learning clause guidance. In: Proceedings of the 10th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, Dagstuhl Publishing (2019)
36. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. *AI Mag.* **33**(1), 89–92 (2012)
37. Kaliszzyk, C., Schulz, S., Urban, J., Vyskočil, J.: System description: E.T. 0.1. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 389–398. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_27
38. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
39. Korovin, K., Kovac, L., Rege, G., J., S., Voronkov, A.: ALASCA: Reasoning in Quantified Linear Arithmetic (Extended Version) (2023). <https://easychair.org/publications/preprint/KJX2>

40. Kotelnikov, E., Kovacs, L., Reger, G., Voronkov, A.: The vampire and the FOOL. In: Avigad, J., Chlipala, A. (eds.) *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 37–48. ACM (2016)
41. Kotthoff, L., Fr chet te, A., Michalak, T., Rahwan, T., Hoos, H., Leyton-Brown, K.: Quantifying algorithmic improvements over time. In: Lang, J. (ed.) *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 5165–5171 (2018)
42. K lwein, D., Blanchette, J.: A survey of axiom selection as a machine learning problem. In: Geschke, S. (ed.) *Computability and Metamathematics: Festschrift Celebrating the 60th birthdays of Peter Koepke and Philip Welch*, pp. 1–15. College Publications (2014)
43. Kumar, R., Myreen, M., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Sewell, P. (ed.) *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 179–191. ACM Press (2014)
44. Loechner, B., Hillenbrand, T.: A phytophany of waldmeister. *AI Commun.* **15**(2/3), 127–133 (2002)
45. Maslej, N., et al.: *The AI Index 2023 Annual Report*. Institute for Human-Centered AI, Stanford University (2023)
46. McCune, W.: Prover9. <http://www.cs.unm.edu/~mccune/prover9/>
47. McCune, W.: Solution of the robbins problem. *J. Autom. Reason.* **19**(3), 263–276 (1997)
48. McCune, W.: Otter 3.3 reference manual. Technical report, ANL/MS-C-TM-263, Argonne National Laboratory, Argonne, USA (2003)
49. McKeown, J., Sutcliffe, G.: Reinforcement learning for guiding the e theorem prover. In: Ae Chun, A., Franklin, M. (eds.) *Proceedings of the 36th International FLAIRS Conference* (2023). <https://doi.org/10.32473/flairs.36.133334>
50. de Moura, L., Bj rner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
51. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.* **1**(2), 245–257 (1979)
52. Ott, S., Barbosa-Silva, A., Blag c, K., Brauner, J., Samwald, M.: Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *Nat. Commun.* **13**(6793), 1–11 (2022)
53. Paulson, L., Blanchette, J.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) *Proceedings of the 8th International Workshop on the Implementation of Logics*, pp. 1–11. No. 2 in *EPiC Series in Computing*, EasyChair Publications (2010)
54. Pelletier, F., Sutcliffe, G., Suttner, C.: The development of CASC. *AI Commun.* **15**(2–3), 79–90 (2002)
55. Peter, L., Hull, R.: *The Peter Principle*. Souvenir Press (1969)
56. Quaife, A.: Automated development of Tarski’s geometry. *J. Autom. Reason.* **5**(1), 97–118 (1989)
57. R mmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) *LPAR 2008*. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20

58. Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_49
59. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
60. Schwabbauser, W., Szmielew, W., Tarski, A.: *Metamathematische Methoden in der Geometrie*. Springer, Heidelberg (1983)
61. Slaney, J., Fujita, M., Stickel, M.: Automated reasoning and exhaustive search: quasigroup existence problems. *Comput. Math. Appl.* **29**(2), 115–132 (1995)
62. Smallbone, N.: Twee: an equational theorem prover. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 602–613. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_35
63. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28
64. Suda, M.: Vampire getting noisy: will random bits help conquer chaos? (system description). In: Blanchette, J., Kovacs, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 659–667. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_38
65. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 6–22. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_4
66. Sutcliffe, G.: The TPTP problem library and associated infrastructure. The FOF and CNF parts, v3.5.0. *J. Autom. Reason.* **43**(4), 337–362 (2009)
67. Sutcliffe, G.: The TPTP world – infrastructure for automated reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 1–12. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_1
68. Sutcliffe, G.: The CADE ATP system competition - CASC. *AI Mag.* **37**(2), 99–101 (2016)
69. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* **59**(4), 483–502 (2017)
70. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formaliz. Reason.* **3**(1), 1–27 (2010)
71. Sutcliffe, G., Desharnais, M.: The 11th IJCAR automated theorem proving system competition - CASC-J11. *AI Commun.* **36**(2), 73–91 (2023)
72. Sutcliffe, G., Desharnais, M.: The CADE-29 automated theorem proving system competition - CASC-29. *AI Commun.* (2024, to appear)
73. Sutcliffe, G., Fuchs, M., Suttner, C.: Progress in automated theorem proving, 1997–2001. In: Hoos, H., Stützle, T. (eds.) *Proceedings of the IJCAI'01 Workshop on Empirical Methods in Artificial Intelligence*, pp. 53–60 (2001)
74. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 406–419. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_32
75. Sutcliffe, G., Suttner, C.: The state of CASC. *AI Commun.* **19**(1), 35–48 (2006)
76. Sutcliffe, G., Suttner, C.: Evaluating general purpose automated theorem proving systems. *Artif. Intell.* **131**(1–2), 39–54 (2001)

77. Sutcliffe, G., Zimmer, J., Schulz, S.: Communication formalisms for automated theorem proving tools. In: Sorge, V., Colton, S., Fisher, M., Gow, J. (eds.) *Proceedings of the Workshop on Agents and Automated Reasoning*, pp. 52–57 (2003)
78. Suttner, C., Sutcliffe, G., Perrault, R.: Technical performance of automated theorem proving (ATP). In: Zhang, D., et al. (eds.) *The AI Index 2021 Annual Report*, pp. 34–35. Human-Centered AI Institute, Stanford University (2021)
79. Suttner, C., Schumann, J.: Parallel automated theorem proving. In: Kanal, L., Kumar, V., Kitano, H., Suttner, C. (eds.) *Parallel Processing for Artificial Intelligence 1*, pp. 209–257. Elsevier Science (1994)
80. Urban, J.: MPTP 0.2: design, implementation, and initial experiments. *J. Autom. Reason.* **37**(1-2), 21–43 (2006)
81. Veroff, R.: A Wos challenge met. *J. Autom. Reason.* **66**, 565–574 (2022)
82. Voronkov, A.: Spider: Learning in the Sea of Options (2023). <https://easychair.org/smart-program/Vampire23/2023-07-05.html>
83. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: Vojnar, T., Zhang, L. (eds.) *TACAS 2019*. LNCS, vol. 11427, pp. 192–210. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_11
84. Wang, H., et al.: LEGO-Prover: Neural Theorem Proving with Growing Libraries (2023). [arXiv:2310.00656](https://arxiv.org/abs/2310.00656)
85. Weidenbach, C., et al.: System description: SPASS version 1.0.0. In: Ganzinger, H. (ed.) *CADE 1999*. LNCS (LNAI), vol. 1632, pp. 378–382. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_34
86. Wos, L.: From the AAR President, Larry Wos. *AAR Newsletter* 129-2019-10 (2019)
87. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: Cimatti, A., Sebastiani, R. (eds.) *SAT 2012*. LNCS, vol. 7317, pp. 228–241. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_18
88. Xu, Y., Liu, J., Chen, S., Zhong, X., He, X.: Contradiction separation based dynamic multi-clause synergized automated deduction. *Inf. Sci.* **462**, 93–113 (2018)
89. Zheng, K., Han, J., Polu, S.: miniF2F: a cross-system benchmark for formal olympiad-level mathematics. In: Liu, Y., Finn, C., Choi, Y., Deisenroth, M. (eds.) *Proceedings of the 10th International Conference on Learning Representations (2022)*



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

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





A Higher-Order Vampire (Short Paper)

Ahmed Bhayat¹  and Martin Suda² 

¹ Leicester, UK

ahmed_bhayat@hotmail.com

² Czech Technical University in Prague, Prague, Czech Republic

martin.suda@cvut.cz

Abstract. The support for higher-order reasoning in the Vampire theorem prover has recently been completely reworked. This rework consists of new theoretical ideas, a new implementation, and a dedicated strategy schedule. The theoretical ideas are still under development, so we discuss them at a high level in this paper. We also describe the implementation of the calculus in the Vampire theorem prover, the strategy schedule construction and several empirical performance statistics.

Keywords: Vampire · Higher-Order · Strategy Scheduling

1 Introduction

The Vampire prover [15] has supported higher-order reasoning since 2019 [6]. Until recently, this support was via a translation from higher-order logic (HOL) to polymorphic first-order logic using combinators. The approach had positives, specifically it avoided the need for higher-order unification. However, our experience suggested that for problems requiring complex unifiers, the approach was not competitive with calculi that do rely on higher-order unification. This intuition was supported by results at the CASC system competition [25].

Due to this, we recently devised an entirely new higher-order superposition calculus. This time we based our calculus on a standard presentation of HOL. The key idea behind our calculus is that rather than using full higher-order unification, we use a depth-bounded version. That is, when searching for higher-order unifiers, when some predefined number of projection and imitation steps have taken place, the search is backtracked. The crucial difference in our approach to similar approaches is that rather than failing on reaching the depth limit, we turn the set of remaining unification pairs into negative constraint literals which are returned along with the substitution formed until that point. This is similar to recent developments in the field of theory reasoning [5].

The new calculus has now been implemented in Vampire along with a dedicated strategy schedule. Together these developments propelled Vampire to first

A. Bhayat—Independent Scholar.

© The Author(s) 2024

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 75–85, 2024.

https://doi.org/10.1007/978-3-031-63498-7_5

place in the THF division of the 2023 edition of the CASC competition.¹ As the completeness of the calculus is an open question which we are working on, we have to date not published a description of the calculus.

In this paper, we describe the calculus, discuss its implementation in Vampire and also provide some details of the strategy schedule and its formation.

2 Preliminaries

We assume familiarity with higher-order logic and higher-order unification. Detailed presentations of these can be found in recent literature [2, 4, 29]

We work with a rank-1 polymorphic, clausal, higher-order logic. For the syntax of the logic we follow a more-or-less standard presentation such as that of Bentkamp et al. [2]. Higher-order applications such as $f a c$ contain subterms with no first-order equivalents such as f and $f a$. We refer to these as *prefix* subterms. We represent term variables with x, y, z , function symbols with f, g, h , and terms with s and t . To keep the presentation simple, we omit typing information from our terms.

A substitution is a mapping of variables to terms. Unification is the process of finding a substitution σ for terms t_1 and t_2 such that $t_1\sigma \approx t_2\sigma$ for some definition of equality (\approx) of interest. It is well known that first-order syntactic unification is decidable and unique most general unifiers exists. For the higher-order case, unification is not decidable, and the set of incomparable unifiers is potentially infinite. A commonly used higher-order unification procedure for enumerating unifiers is Huet's preunification routine [13]. Unlike full higher-order unification, preunification does not attempt to unify terms if both have variable head symbols. Thus, preunification does not require infinitely branching rules unlike full higher-order unification [29].

The two main rules that extend first-order unification in Huet's procedure are *projection* and *imitation*. We provide a flavour of these via an example. Consider unifying terms $s = x a$ and $s' = a$. In searching for a suitable instantiation of the variable x , we can either attempt to copy the head symbol of s' leading to the substitution $x \rightarrow \lambda y. a$, or we can bring one of x 's arguments to the head position leading to the substitution $x \rightarrow \lambda y. y$. The first is known as imitation and the second as projection.

We use the concept of a *depth_n unifier*. We do not define the term formally, but provide an intuitive understanding. Consider a higher-order preunification algorithm. Any substitution formed by following a path of the unification tree, starting from the root, that contains exactly n imitation and projection steps, or reaches a leaf using fewer than n such steps, is a *depth_n unifier*. For terms s and t , let $U_n(s, t)$ be the set of all depth_n unifiers of s and t . Note that this set is finite as we are assuming preunification and hence the tree is finitely branching.

For terms s and t , for each depth_n unifier $\sigma \in U_n(s, t)$, we associate a set of negative equality literals C_σ formed by turning the unification pairs that remain

¹ <https://tptp.org/CASC/29/WWWFiles/DivisionSummary1.html>.

when the depth limit is reached into negative equalities. In the case σ is an *actual unifier* of s and t , C_σ is of course the empty set.

To make this clearer, consider the unification tree presented in Fig. 1. There are two depth₂ unifiers labelled σ_1 and σ_2 in the figure. Related to these, we have $C_{\sigma_1} = C_{\sigma_2} = \{x_2 a b \not\approx b\}$. There are four depth₃ unifiers (not shown in the figure) and zero depth _{n} unifiers for for $n > 3$.

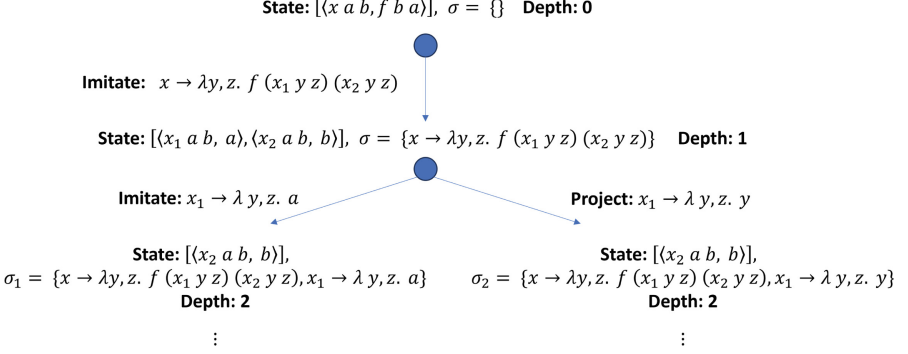


Fig. 1. Unification tree for terms $x a b$ and $f b a$

3 Calculus

Our calculus is parameterised by a selection function and an ordering \succ . Together these give rise to the concept of literals being (strictly) \succ -eligible with respect to a substitution σ [2]. When discussing eligibility we drop \succ and σ and rely on the context to make these clear. We call a literal $s \not\approx t$, where both s and t have variable heads, a *flex-flex* literal. Such a literal is never selected in the calculus. We present the primary inference rule, SUP, below.

$$\frac{D' \vee t \approx t' \quad C' \vee s \langle u \rangle \approx s'}{(C' \vee D' \vee s \langle t' \rangle \approx s' \vee C_\sigma) \sigma} \text{ SUP}$$

In the rule above, we use \approx to denote either a positive or negative equality. We use $s \langle u \rangle$ to denote that u is a *first-order* subterm of s . That is, a non-prefix subterm that is not below a lambda. The side conditions of the inference are $\sigma \in U_n(t, u)$, u is not a variable, $t \approx t'$ is strictly eligible in the left premise, $s \langle u \rangle \approx s'$ is eligible in the right premise, and the other standard ordering conditions. The remaining core inference rules are EQRES and EQFACT.

$$\frac{C' \vee t \approx t' \vee s \approx s'}{(C' \vee t' \not\approx s' \vee s \approx s' \vee C_\sigma) \sigma} \text{ EQFACT} \qquad \frac{C' \vee s \not\approx t'}{(C' \vee C_\sigma) \sigma} \text{ EQRES}$$

For both rules, $\sigma \in U_n(t, s)$. For EQFACT, $s \approx s'$ is eligible in the premise and for EQRES $s \not\approx s'$ is eligible. We also include inferences ARGCONG (see [3]), and FLEXFLEXSIMP which derives the empty clause, \perp , from a clause containing only flex-flex literals.

$$\frac{C' \vee s \approx s'}{C' \sigma \vee (s\sigma) x \approx (s'\sigma) x} \text{ ARGCONG} \quad \frac{x_1 \bar{s}_n \not\approx x_2 \bar{r}_m \vee \dots}{\perp} \text{ FLEXFLEXSIMP}$$

For ARGCONG, $s \approx s'$ is eligible in the premise, σ is the type unifier of s and s' and x is a fresh variable. In our implementation, the depth parameter n is set via a user option. In the case it is set to 0, the following pair of inferences are added to the calculus.

$$\frac{C' \vee x \bar{s}_n \not\approx f \bar{t}_m}{(C' \vee x \bar{s}_n \not\approx f \bar{t}_m)\{x \rightarrow \lambda \bar{y}_n. f(z_j \bar{y}_n)_m\}} \text{ IMITATE}$$

$$\frac{C' \vee x \bar{s}_n \not\approx f \bar{t}_m}{(C' \vee x \bar{s}_n \not\approx f \bar{t}_m)\{x \rightarrow \lambda \bar{y}_n. y_i(z_j \bar{y}_n)_p\}} \text{ PROJECT}$$

Where j ranges from 1 to m in IMITATE and 1 to p in PROJECT, and each z_j is a fresh variable. The literals $x \bar{s}_n \not\approx f \bar{t}_m$ are eligible in the premises and p is the arity of y_i , the projected variable. The idea behind introducing these rules is to facilitate the instantiation of head variables with suitable lambda terms when this is not being done as part of unification. Our intuition is that by intertwining the unification and calculus rules in the spirit of the EP calculus [21], the need for explosive rules (such as FLUIDSUP [2]) that simulate superposition underneath variables is removed. The examples we present below support this intuition. Besides the core inference rules, the calculus has a set of rules to handle reasoning about Boolean terms. These are similar to rules discussed in the literature [20, 30]. Extensionality is supported either via an axiom or by using unification with abstraction as described by Bhayat [4]. Similarly, Hilbert choice can be supported via a lightweight inference in the manner of Leo-III [20] or via the addition of the Skolemized choice axiom. The calculus also contains various well-known simplification rules such as DEMODULATION and SUBSUMPTION.

Soundness and Completeness. The soundness of the calculus described above is relatively straightforward to show. On the other hand, the completeness of the calculus with respect to Henkin semantics is an open question. We hypothesise that given the right ordering, and with tuning of inference side conditions, the depth₀ variant of the calculus (with the IMITATE and PROJECT rules) is refutationally complete. A proof is unlikely to be straightforward due to the fact that we do not select flex-flex literals.

Example 1. Consider the following unsatisfiable clause set. Assume a depth of 1. Selected literals are underlined.

$$C = x a b \not\approx \underline{f b a} \vee x c d \not\approx \underline{f b a}$$

An EQRES binds x to $\lambda y, z. f(x_1 a b)(x_2 a b)$ and results in $C_1 = f(x_1 a b)(x_2 a b) \not\approx f b a \vee f(x_1 c d)(x_2 c d) \not\approx f b a$. An EQRES on C_1 binds x_1 to $\lambda y, z. b$ and results in $C_2 = x_2 a b \not\approx a \vee f b(x_2 c d) \not\approx f b a$. A final EQRES on C_2 binds x_2 to $\lambda y, z. a$ and results in $\underline{f b a} \not\approx f b a$ from which it is trivial to obtain the empty clause \perp .

Example 2 (Example 1 of Bentkamp et al. [3]). Consider the following unsatisfiable clause set. Assume the depth_0 version of the calculus.

$$C_1 = f a \approx c \quad C_2 = h(yb)(ya) \not\approx h(g(fb))(gc)$$

An EQRES inference on C_2 results in $C_3 = yb \not\approx g(fb) \vee ya \not\approx gc$. An IMITATE inference on the first literal of C_3 followed by the application of the substitution and some β -reduction results in $C_4 = g(zb) \not\approx g(fb) \vee g(za) \not\approx gc$. A further double application of EQRES gives us $C_5 = zb \not\approx fb \vee za \not\approx c$. We again carry out IMITATE on the first literal followed by an EQRES to leave us with $C_6 = xb \not\approx b \vee f(xa) \not\approx c$. We can now carry out a SUP inference between C_1 and C_6 resulting in $C_7 = xb \not\approx b \vee c \not\approx c \vee xa \not\approx a$ from which it is simple to derive \perp via an application of IMITATE on either the first or the third literal. Note, that the empty clause was derived without the need for an inference that simulates superposition underneath variables, unlike in [3].

4 Implementation

The calculus described above, along with a dedicated strategy schedule, has been implemented in the Vampire theorem prover.² Vampire natively supports rank-1 polymorphic first-order logic. Therefore, we translate higher-order terms into polymorphic first-order terms using the well known applicative encoding. Note, that we use the symbol \mapsto , in a first-order type, to separate the argument types from the return type. It should not be confused with the binary, higher-order function type constructor \rightarrow that we assume to be in the type signature. Application is represented by a polymorphic symbol $app : \Pi\alpha_1, \alpha_2. (\alpha_1 \rightarrow \alpha_2 \times \alpha_1) \mapsto \alpha_2$. Lambda terms are stored internally using De Bruijn indices. A lambda is represented by a polymorphic symbol $lam : \Pi\alpha_1, \alpha_2. \alpha_2 \mapsto (\alpha_1 \rightarrow \alpha_2)$. De Bruijn indices are represented by a family of polymorphic symbols $d_i : \Pi\alpha. \alpha$ for $i \in \mathbb{N}$. Thus, the term $\lambda x : \tau. x$ is represented internally as $lam(\tau, \tau, d_0(\tau))$. The term $\lambda x. f(\lambda z. x)$ is represented internally (now ignoring type arguments) as $lam(app(f, lam(d_1)))$.

Some of the most important options available are: `hol_unif_depth` to control the depth unification proceeds to, `funx_ext` to control how function extensionality is handled, `cnf_on_the_fly` to control how eager or lazy the clausification algorithm is, and `applicative_unif` which replaces higher-order unification with (applicative) first-order unification. This is surprisingly helpful in some cases. Besides for the options listed above, there are many other higher-order specific options as well as options that impact both higher-order and first-order reasoning. These options can be viewed by building Vampire and running with `-help`.

² See <http://bit.ly/3vBQLi4> for the release, <https://bit.ly/3HI3IES> for the code.

5 Strategies and the Schedule

We generally followed the Spider [27] methodology for strategy discovery and schedule creation. This starts with randomly sampling strategies to solve as-of-yet unsolved problems (or improve the best known time for problems already known to be solvable). Each newly discovered strategy is optimized with local search to work even better on the single problem which it just solved. This is done by trying out alternative values for each option, possibly in several rounds. A variant of the strategy that improves the solution time or at least uses a default value of an option is preferred. The final strategy is then evaluated on the totality of all considered problems and the process repeats.

In our case, we sought strategies to cover the 3914 TH0 problems of the TPTP library [24] version 8.1.2. The strategy space consisted of 87 options inherited from first-order Vampire and 26 dedicated higher-order options. To sample a random strategy, we considered each option separately and picked its value based on a guess of how useful each is. (E.g., for `applicative_unif` we used the relative frequencies of `on`: 3, `off`: 10.) During the strategy discovery process we adapted the maximum running time per problem, both for the random probes several times and for the final strategy evaluation: from the order of 1s up to 100s. In total, we collected 1158 strategies over the course of approximately two weeks of continuous 60 core CPU computation. The strategies cover 2804 unsatisfiable problems, including 50 problems of TPTP rating 1.0 (which means these problems were not officially solved by an ATP before).

Once a sufficiently rich set of strategies gets discovered and evaluated, schedule building can be posed as a constraint programming task in which one seeks to allot time slices to individual strategies to cover as many problems as possible while not exceeding a given overall time bound T [12,19]. We had a good experience with a weighted set cover formulation and applying a greedy algorithm [9]: starting from an empty schedule, at any point we decide to extend it by scheduling a strategy S for additional t units of time if this step is currently the best among all possible strategy extensions in terms of “the number of problems that will additionally get covered *divided by* t ”. This greedy approach does not guarantee an optimal result, but runs in polynomial time and gives a meaningful answer uniformly for any overall time bound T (See [8] for more details).

Our final higher-order schedule tries to cover, in this greedy sense, as many problems as possible at several increasing time bounds: starting from 1s, 5s, and 10s bounds relevant for the impatient users, all the way up to the CASC limit of 16 min (2 min on 8 cores) and further beyond. In the end, it makes use of 278 out of the 1158 available strategies and manages to cover all the known-to-be-solvable problems in a bit less than 1 h of single core computation. We stress that our final schedule is a single monolithic sequence and does not branch based on any problems’ characteristics or features.³

³ One additional interesting aspect of our schedule building approach (see Appendix A of our preprint [7] for more details) is that we employ input shuffling and prover randomization [23] and thus treat our strategies as Las Vegas algorithms, whose running time or even success/failure may depend on chance.

Table 1. The most important options in terms of contribution to problem coverage

an option	default	# problems not solvable without non-default
<code>cnf_on_the_fly</code>	<code>eager</code>	102
<code>applicative_unif</code>	<code>off</code>	56
<code>equality_to_equiv</code>	<code>off</code>	24
<code>hol_unif_depth</code>	2	20
<code>func_ext</code>	<code>abstraction</code>	12

Most Important Options: In Table 1, we list the first five options sorted in descending order of “how many problems we would not be able to cover if the given option could not be varied in strategies.” (In other words, as if the listed default value was “wired-in” to the prover code.)

Based on existing research [28], it is unsurprising to see that varying clausification has a large impact. Likewise, for varying the unification depth. What is perhaps more surprising is that replacing higher-order unification with applicative first-order unification can be beneficial. `equality_to_equiv` turns equality between Boolean terms into equivalence before the original clausification pass is carried out. The effectiveness of this option is also somewhat surprising.

Table 2. Number of problems solved by a single good higher-order strategy and our schedule at various time limit cutoffs. Run on the 3914 TH0 TPTP problems

	1 s	10 s	30 s	60 s	120 s	960 s
single strategy	1811	1949	2041	2094	—	—
our schedule	2067	2436	2584	2642	2691	2775

Performance Statistics: It is long known [26,31] that a strategy schedule can improve over the performance of a single good strategy by large margin. Table 2 confirms this phenomenon for our case. For this comparison we selected one of the best performing (at the 60 s time limit mark) single strategies that we had previously evaluated. From the higher-order perspective, the strategy is interesting for setting `hol_unif_depth` to 4 and supporting choice reasoning via an inference rule (`choice_reasoning on`).⁴

Although our schedule has been developed on (and for) the TH0 TPTP problems, it helps the new higher-order Vampire solve more problems of other origin too. Of the Sledgehammer problems exported by Desharnais et al. in their

⁴ Otherwise, it uses Vampire’s default setting, except for relying on an incomplete literal selection function [11] and using a relative high naming threshold [17], i.e., being reluctant to introduce new names for subformulas during clausification.

last prover comparison [10], namely the 5000 problems denoted in their work TH0⁻, Vampire can now solve 2425 compared to 2179 obtained by Desharnais et al. with the previous Vampire version (both under 30s per problem).⁵

We remark that we also developed a different schedule specifically adapted to Sledgehammer problems (in various TPTP dialects, i.e., not just TH0), which is now available to the Isabelle [16] users since the September 2023 release.

6 Related Work

The idea to intertwine superposition and unification appears in earlier work, particularly in the EP calculus implemented in Leo-III [21]. The main differences between our calculus and EP are:

1. We do not move first-order unification to the calculus level. Hence, there are no equivalents to the TRIV, BIND and DECOMP rules of EP.
2. Our PROJECT and IMITATE rules are instances of EP’s FLEXRIGID rule. We do not include an equivalent to EP’s FLEXFLEX rule since we never select flex-flex literals. Instead, we leave such literals until one of the head variables becomes instantiated, or the clause only contains flex-flex literals at which point FLEXFLEXSIMP can be applied.
3. Our core inference rules are parameterised by a selection function and an ordering.
4. Whilst EP always applies unification lazily, our calculus can control how lazily unification is carried out by varying the depth bound.⁶

We also incorporate more recent work on higher-order superposition, mainly from the Matryoshka project [2, 28]. Of course, the use of constraints in automated reasoning extends far beyond the realm of higher-order logic. They have been researched in the context of theory reasoning [14, 18] and basic superposition [1].

7 Conclusion

In this paper, we have presented a new higher-order superposition calculus and discussed its implementation in Vampire. We have also described the new higher-order schedule created. The combination of calculus, implementation and schedule have already proven effective. However, we believe that there is great room for further exploration and improvement. On the theoretical side, we wish to prove refutational completeness of the calculus (or a variant thereof). On the practical side, we wish to refine the implementation, most notably by adding additional simplification rules.

⁵ Our experiments were run on Intel®Xeon®Gold 6140 CPU @ 2.3 GHz, Desharnais et al. [10] used StarExec [22] with Intel®Xeon®CPU E5-2609 @ 2.4 GHz nodes.

⁶ Our understanding is that the implementation of EP in Leo-III does make use of orderings as well as eager unification. However, eager unification does not return unification literals, instead failing once the depth bound is reached. See [20] for details.

Acknowledgments. The second author was supported by project CORESENSE no. 101070254 under the Horizon Europe programme and project RICAIP no. 857306 under the EU-H2020 programme.

References

1. Bachmair, L., Ganzinger, H., Lynch, C., Snyder, W.: Basic paramodulation and superposition. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 462–476. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_185
2. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for higher-order logic. *J. Autom. Reason.* **67**(1), 10 (2023)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 55–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_4
4. Bhayat, A.: Automated theorem proving in higher-order logic. Ph.D. thesis (2015)
5. Bhayat, A., Korovin, K., Kovács, L., Schoisswohl, J.: Refining unification with abstraction. In: LPAR, pp. 36–47 (2023)
6. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 278–296. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_16
7. Bhayat, A., Suda, M.: A higher-order vampire (short paper). EasyChair Preprint no. 13125 (EasyChair, 2024)
8. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction. In: IJCAR (2024, accepted)
9. Chvátal, V.: A greedy heuristic for the set-covering problem. *Math. Oper. Res.* **4**(3), 233–235 (1979)
10. Desharnais, M., Vukmirović, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer. In: ITP. LIPIcs, vol. 237, pp. 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
11. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 313–329. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_22
12. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) CICM 2021. LNCS (LNAI), vol. 12833, pp. 107–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81097-9_8
13. Huet, G.P.: A unification algorithm for typed λ -calculus. *Theoret. Comput. Sci.* **1**(1), 27–57 (1975)
14. Korovin, K., Kovács, L., Reger, G., Schoisswohl, J., Voronkov, A.: ALASCA: Reasoning in quantified linear arithmetic. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS. LNCS, vol. 13993, pp. 647–665. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_33
15. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant For Higher-order Logic, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>

17. Regeer, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: GCAI. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016)
18. Regeer, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 3–22. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_1
19. Schurr, H.: Optimal strategy schedules for everyone. In: PAAR. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022)
20. Steen, A.: Extensional paramodulation for higher-order logic and its effective implementation Leo-III. Ph.D. thesis (2018)
21. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. *J. Autom. Reason.* **65**(6), 775–807 (2021)
22. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28
23. Suda, M.: Vampire getting noisy: will random bits help conquer chaos? (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 659–667. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_38
24. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* **59**(4), 483–502 (2017)
25. Sutcliffe, G., Suttner, C.: The state of CASC. *AI Commun.* **19**, 35–48 (2006)
26. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS, vol. 1421, pp. 427–441. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054276>
27. Voronkov, A.: Spider: learning in the sea of options. In: Vampire23: The 7th Vampire Workshop (2023, to appear). <https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833>
28. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 415–432. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_24
29. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. *Logical Methods in Computer Science* **17** (2021)
30. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: PAAR, pp. 148–166 (2020)
31. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) FLAIRS, pp. 142–146. AAAI Press (1998)



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

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





Tableaux for Automated Reasoning in Dependently-Typed Higher-Order Logic

Johannes Niederhauser¹(✉) , Chad E. Brown², and Cezary Kaliszyk^{1,3} 

¹ Department of Computer Science, University of Innsbruck, Innsbruck, Austria
johannes.niederhauser@uibk.ac.at, cezarykaliszyk@gmail.com

² Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical
University in Prague, Prague, Czech Republic

³ School of Computing and Information Systems, University of Melbourne,
Melbourne, Australia

Abstract. Dependent type theory gives an expressive type system facilitating succinct formalizations of mathematical concepts. In practice, it is mainly used for interactive theorem proving with intensional type theories, with PVS being a notable exception. In this paper, we present native rules for automated reasoning in a dependently-typed version (DHOL) of classical higher-order logic (HOL). DHOL has an extensional type theory with an undecidable type checking problem which contains theorem proving. We implemented the inference rules as well as an automatic type checking mode in Lash, a fork of Satallax, the leading tableaux-based prover for HOL. Our method is sound and complete with respect to provability in DHOL. Completeness is guaranteed by the incorporation of a sound and complete translation from DHOL to HOL recently proposed by Rothgang et al. While this translation can already be used as a preprocessing step to any HOL prover, to achieve better performance, our system directly works in DHOL. Moreover, experimental results show that the DHOL version of Lash can outperform all major HOL provers executed on the translation.

Keywords: Tableaux · Dependent Types · Higher-Order Logic

1 Introduction

Dependent types introduce the powerful concept of types depending on terms. Lists of fixed length are an easy but interesting example. Instead of having a simple type `lst` we may have a type $\Pi n : \text{nat}. \text{lst } n$ which takes a natural number as argument and returns the type of a list with length n . More generally, lambda terms $\lambda x. s$ now have a dependent type $\Pi x : A. B$ which makes the type of $(\lambda x. s) t$ dependent on t . With that, it is possible for example to specify an unfailling version of the tail function by declaring its type to be $\Pi n : \text{nat}. \text{lst}(sn) \rightarrow \text{lst } n$. Many interactive theorem provers for dependent type theory are available [3, 10, 14, 16], most of them implement intensional type theories, i.e., they distinguish between a decidable judgmental equality (given by conversions) and provable equality (inhabiting an identity type). Notable exceptions are PVS [19] and F* [21] which

implement an extensional type theory. In the context of this paper, we say a type theory is *extensional* if judgmental equality and provable equality coincide, as in [12]. The typing judgment in such type theories is usually undecidable, as shown in [8].

The broader topic of this paper is automated reasoning support for extensional type theories with dependent types. Not much has been done to this end, but last year Rothgang et al. [17] introduced an extension of HOL to dependent types which they dub DHOL. In contrast to dependent type theory, automated theorem proving in HOL has a long history and led to the development of sophisticated provers [2, 4, 20]. Rothgang et al. defined a natural extension of HOL and equipped it with automation support by providing a sound and complete translation from DHOL into HOL. Their translation has been implemented and can be used as a preprocessing step to any HOL prover in order to obtain an automated theorem prover for DHOL. Hence, by committing to DHOL, automated reasoning support for extensional dependent type theories does not have to be invented from scratch but can benefit from the achievements of the automated theorem proving community for HOL.

In this paper, we build on top of the translation from Rothgang et al. to develop a tableau calculus which is sound and complete for DHOL. In addition, dedicated inference rules for DHOL are defined and their soundness is proved. The tableau calculus is implemented as an extension of Lash [6]. The remainder of this paper is structured as follows: Sect. 2 sets the stage by defining DHOL and the erasure from DHOL to HOL due to Rothgang et al. before Sect. 3 defines the tableau calculus and provides soundness and completeness proofs. The implementation is described in Sect. 4. Finally, we report on experimental results in Sect. 5.

2 Preliminaries

2.1 HOL

We start by giving the syntax of higher-order logic (HOL) which goes back to Church [9]. In order to allow for a graceful extension to DHOL, we define it with a grammar based on [17].

$$\begin{aligned}
 T &::= \circ \mid T, a: \text{tp} \mid T, x: A \mid T, s && \text{(theories)} \\
 \Gamma &::= \cdot \mid \Gamma, x: A \mid \Gamma, s && \text{(contexts)} \\
 A, B &::= a \mid A \rightarrow B \mid o && \text{(types)} \\
 s, t, u, v &::= x \mid \lambda x: A. s \mid s t \mid \perp \mid \neg s \mid s \Rightarrow t \mid s =_A t \mid \forall x: A. s && \text{(terms)}
 \end{aligned}$$

A theory consists of base type declarations $a: \text{tp}$, typed variable or constant declarations $x: A$ and axioms. Contexts are like theories but without base type declarations. In the following, we will often write $s \in T, \Gamma$ to denote that s occurs in the combination of T and Γ . Furthermore, note that \circ and \cdot denote the empty theory and context, respectively. Types are declared base types a ,

the base type of booleans o or function types $A \rightarrow B$. As usual, the binary type constructor \rightarrow is right-associative. Terms are simply-typed lambda-terms (modulo α -conversion) enriched by the connectives \perp , \neg , \Rightarrow , $=_A$ as well as the typed binding operator for \forall . All connectives yield terms of type o (formulas). By convention, application associates to the left, so $st u$ means $(s t) u$ with the exception that $\neg s t$ always means $\neg(s t)$. Moreover, we abbreviate $\neg(s =_A t)$ by $s \neq_A t$ and sometimes omit the type subscript of $=_A$ when it is either clear from the context or irrelevant. We write $s[x_1/t_1, \dots, x_n/t_n]$ to denote the simultaneous capture-avoiding substitution of the x_i 's by the t_i 's. The set of free variables of a term s is denoted by $\mathcal{V}s$.

A theory T is well-formed if all types are well-formed and axioms have type o with respect to its base type declarations. In that case, we write $\vdash^s T$ Thy where the superscript s indicates that we are in the realm of simple types. Given a well-formed theory T , the well-formedness of a context Γ is defined in the same way and denoted by $\vdash_T^s \Gamma$ Ctx. Given a theory T and a context Γ , we write $\Gamma \vdash_T^s A$ tp to state that A is a well-formed type and $\Gamma \vdash_T^s s : A$ to say that s has type A . Furthermore, $\Gamma \vdash_T^s s$ denotes that s has type o and is provable from Γ and T in HOL. Finally, we use $\Gamma \vdash_T^s A \equiv B$ to state that A and B are equivalent well-formed types. For HOL this is trivial as it corresponds to syntactic equivalence, but this will change drastically in DHOL.

2.2 DHOL

The extension from HOL to DHOL consists of two crucial ingredients:

- the type constructor $A \rightarrow B$ is replaced by the constructor $\Pi x : A. B$ which potentially makes the return type B dependent on the actual argument x ; we stick to the usual arrow notation if B does not contain x
- base types a can now take term arguments; for an n -ary base type we write $a : \Pi x_1 : A_1. \dots \Pi x_n : A_n. \text{tp}$

Thus, the grammar defining the syntax of DHOL is given as follows:

$$\begin{aligned}
 T &::= \circ \mid T, a : (\Pi x : A.)^* \text{tp} \mid T, x : A \mid T, s && \text{(theories)} \\
 \Gamma &::= \cdot \mid \Gamma, x : A \mid \Gamma, s && \text{(contexts)} \\
 A, B &::= a t_1 \dots t_n \mid \Pi x : A. B \mid o && \text{(types)} \\
 s, t, u, v &::= x \mid \lambda x : A. s \mid s t \mid \perp \mid \neg s \mid s \Rightarrow t \mid s =_A t \mid \forall x : A. s && \text{(terms)}
 \end{aligned}$$

If a base type a has arity 0, it is called a *simple base type*. Note that HOL is the fragment of DHOL where all base types have arity 0. Allowing base types to have term arguments makes type equality a highly non-trivial problem in DHOL. For example, if $\Gamma \vdash_T^d s : \Pi x : A. B$ (the d in \vdash^d indicates that we are speaking about DHOL) and $\Gamma \vdash_T^d t : A'$ we still want $\Gamma \vdash_T^d (s t) : B[x/t]$ to hold if $\Gamma \vdash_T^d A \equiv A'$, so checking whether two types are equal is a problem which occurs frequently in DHOL. Intuitively, we have $\Gamma \vdash_T^d A \equiv A'$ if and only if their simply-typed skeleton consisting of arrows and base types without

their arguments is equal and given a base type $a: \Pi x_1: A_1. \dots \Pi x_n: A_n. \mathbf{tp}$, an occurrence $a t_1 \dots t_n$ in A and its corresponding occurrence $a t'_1 \dots t'_n$ in A' , we have $\Gamma \vdash_T^d t_i =_{A_i[x_1/t_1, \dots, x_{i-1}/t_{i-1}]} t'_i$ for all $1 \leq i \leq n$. This makes DHOL an extensional type theory where already type checking is undecidable as it requires theorem proving. Another difference from HOL is the importance of the chosen representation of contexts and theories: Since the well-typedness of a term may depend on other assumptions, the order of the type declarations and formulas in a context Γ or theory T is relevant. A formal definition of the judgments $\Gamma \vdash_T^d A \mathbf{tp}$, $\Gamma \vdash_T^d s: A$, $\Gamma \vdash_T^d s$ and $\Gamma \vdash_T^d A \equiv B$ via an inference system is given in [17]. Since we use more primitive connectives, a minor variant is presented in Fig. 1.

Example 1. Consider the simple base types $\mathbf{nat}: \mathbf{tp}$ and $\mathbf{elem}: \mathbf{tp}$ as well as the dependent base type $\mathbf{lst}: \Pi x: \mathbf{nat}. \mathbf{tp}$. The constants and functions

$$\begin{array}{ll} 0: \mathbf{nat} & \mathbf{s}: \mathbf{nat} \rightarrow \mathbf{nat} \\ \mathbf{nil}: \mathbf{lst} \ 0 & \mathbf{cons}: \Pi n: \mathbf{nat}. \mathbf{elem} \rightarrow \mathbf{lst} \ n \rightarrow \mathbf{lst} \ (\mathbf{s} \ n) \end{array}$$

provide means to represent their inhabitants. Additionally, we define functions $\mathbf{plus}: \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}$

$$\forall n: \mathbf{nat}. \mathbf{plus} \ 0 \ n =_{\mathbf{nat}} n \quad \forall n, m: \mathbf{nat}. \mathbf{plus} \ (\mathbf{s} \ n) \ m =_{\mathbf{nat}} \mathbf{s} \ (\mathbf{plus} \ n \ m)$$

and $\mathbf{app}: \Pi n: \mathbf{nat}. \Pi m: \mathbf{nat}. \mathbf{lst} \ n \rightarrow \mathbf{lst} \ m \rightarrow \mathbf{lst} \ (\mathbf{plus} \ n \ m)$:

$$\begin{array}{l} \forall n: \mathbf{nat}, x: \mathbf{lst} \ n. \mathbf{app} \ 0 \ n \ \mathbf{nil} \ x =_{\mathbf{lst} \ n} x \\ \forall n, m: \mathbf{nat}, z: \mathbf{elem}, x: \mathbf{lst} \ n, y: \mathbf{lst} \ m. \\ \mathbf{app} \ (\mathbf{s} \ n) \ m \ (\mathbf{cons} \ n \ z \ x) \ y =_{\mathbf{lst} \ (\mathbf{s} \ (\mathbf{plus} \ n \ m))} \mathbf{cons} \ (\mathbf{plus} \ n \ m) \ z \ (\mathbf{app} \ n \ m \ x \ y) \end{array}$$

In the defining equations of \mathbf{app} , we annotated the equality sign with the dependent type of the term on the right-hand side. In all cases, the simply-typed skeleton is just \mathbf{lst} but for a type check we need to prove the two equalities

$$\forall n: \mathbf{nat}. \mathbf{plus} \ 0 \ n =_{\mathbf{nat}} n \quad \forall n, m: \mathbf{nat}. \mathbf{plus} \ (\mathbf{s} \ n) \ m =_{\mathbf{nat}} \mathbf{s} \ (\mathbf{plus} \ n \ m)$$

which are exactly the corresponding axioms for \mathbf{plus} . Type checking the conjecture

$$\forall n: \mathbf{nat}, x: \mathbf{lst} \ n. \mathbf{app} \ n \ 0 \ x \ \mathbf{nil} =_{\mathbf{lst} \ n} x$$

would require proving $\forall n: \mathbf{nat}. \mathbf{plus} \ n \ 0 =_{\mathbf{nat}} n$ which can be achieved by induction on natural numbers if we include the Peano axioms.

2.3 Erasure

The following definition presents the translation from DHOL to HOL due to Rothgang et al. [17]. Intuitively, the translation erases dependent types to their simply typed skeletons by ignoring arguments of base types. The thereby lost

$$\begin{array}{c}
\frac{}{\vdash^d \circ \text{Thy}} \text{thyEmpty} \quad \frac{\vdash_T^d x_1 : A_1, \dots, x_n : A_n \text{ Ctx}}{\vdash^d T, a : \prod x_1 : A_1. \dots \prod x_n : A_n. \text{tp Thy}} \text{thyType} \\
\frac{\vdash_T^d A \text{ tp}}{\vdash^d T, x : A \text{ Thy}} \text{thyConst} \quad \frac{\vdash_T^d s : o}{\vdash^d T, s \text{ Thy}} \text{thyAxiom} \quad \frac{\vdash^d T \text{ Thy}}{\vdash_T^d \cdot \text{Ctx}} \text{ctxEmpty} \\
\frac{\Gamma \vdash_T^d A \text{ tp}}{\vdash_T^d \Gamma, x : A \text{ Ctx}} \text{ctxVar} \quad \frac{\Gamma \vdash_T^d s : o}{\vdash_T^d \Gamma, s \text{ Ctx}} \text{ctxAssume} \\
\frac{a : (\dots \prod x_i : A_i. \dots \text{tp}) \in T \quad \vdash_T^d \Gamma \text{ Ctx} \quad \dots \Gamma \vdash_T^d t_i : A_i[x_1/s_1 \dots x_{i-1}/s_{i-1}] \dots}{\Gamma \vdash_T^d a t_1 \dots t_n \text{ tp}} \text{type} \\
\frac{x : A' \in T \quad \Gamma \vdash_T^d A' \equiv A}{\Gamma \vdash_T^d x : A} \text{const} \quad \frac{s \in T \quad \vdash_T^d \Gamma \text{ Ctx}}{\Gamma \vdash_T^d s} \text{axiom} \\
\frac{x : A' \in \Gamma \quad \Gamma \vdash_T^d A' \equiv A}{\Gamma \vdash_T^d x : A} \text{var} \quad \frac{s \in \Gamma \quad \vdash_T^d \Gamma \text{ Ctx}}{\Gamma \vdash_T^d s} \text{assume} \quad \frac{\vdash_T^d \Gamma \text{ Ctx}}{\Gamma \vdash_T^d o \text{ tp}} \text{bool} \\
\frac{\Gamma \vdash_T^d A \text{ tp} \quad \Gamma, x : A \vdash_T^d B \text{ tp}}{\Gamma \vdash_T^d \prod x : A. B \text{ tp}} \text{pi} \quad \frac{\Gamma \vdash_T^d A \equiv A' \quad \Gamma, x : A \vdash_T^d B \equiv B'}{\Gamma \vdash_T^d \prod x : A. B \equiv \prod x : A'. B'} \text{congII} \\
\frac{a : (\dots \prod x_i : A_i. \dots \text{tp}) \in T \quad \vdash_T^d \Gamma \text{ Ctx} \quad \dots \Gamma \vdash_T^d s_i =_{A_i[x_1/s_1 \dots x_{i-1}/s_{i-1}]} t_i \dots}{\Gamma \vdash_T^d a s_1 \dots s_n \equiv a t_1 \dots t_n} \text{congBT} \\
\frac{\Gamma, x : A \vdash_T^d t : B}{\Gamma \vdash_T^d (\lambda x : A. t) : \prod x : A. B} \text{lambda} \quad \frac{\Gamma \vdash_T^d s : \prod x : A. B \quad \Gamma \vdash_T^d t : A}{\Gamma \vdash_T^d (s t) : B[x/t]} \text{appl} \\
\frac{\Gamma \vdash_T^d s : A \quad \Gamma \vdash_T^d t : A}{\Gamma \vdash_T^d (s =_A t) : o} \text{=type} \quad \frac{\Gamma \vdash_T^d A \equiv A' \quad \Gamma, x : A \vdash_T^d t =_B t'}{\Gamma \vdash_T^d \lambda x : A. t =_{\prod x : A. B} \lambda x : A'. t'} \text{cong}\lambda \\
\frac{\Gamma \vdash_T^d s =_{\prod x : A. B} s' \quad \Gamma \vdash_T^d t =_A t'}{\Gamma \vdash_T^d s t =_{B[x/t]} s' t'} \text{congAppI} \quad \frac{\Gamma \vdash_T^d s : A}{\Gamma \vdash_T^d s =_A s} \text{refl} \\
\frac{\Gamma \vdash_T^d s =_A t}{\Gamma \vdash_T^d t =_A s} \text{sym} \quad \frac{\Gamma \vdash_T^d s : (\prod x : A. B) \quad x \notin \mathcal{V}s}{\Gamma \vdash_T^d s =_{\prod x : A. B} \lambda x : A. s x} \text{eta} \\
\frac{\Gamma \vdash_T^d (\lambda x : A. s) t : B}{\Gamma \vdash_T^d (\lambda x : A. s) t =_B s[x/t]} \text{beta} \quad \frac{\vdash_T^d \Gamma \text{ Ctx}}{\Gamma \vdash_T^d \perp : o} \perp \text{type} \quad \frac{\Gamma \vdash_T^d s : o \quad \Gamma \vdash_T^d \perp}{\Gamma \vdash_T^d s} \perp \text{e} \\
\frac{\Gamma \vdash_T^d s : o}{\Gamma \vdash_T^d (\neg s) : o} \neg \text{type} \quad \frac{\Gamma \vdash_T^d s : o \quad \Gamma, s \vdash_T^d \perp}{\Gamma \vdash_T^d \neg s} \neg \text{i} \quad \frac{\Gamma \vdash_T^d s \quad \Gamma \vdash_T^d \neg s}{\Gamma \vdash_T^d \perp} \neg \text{e} \\
\frac{\Gamma \vdash_T^d \neg \neg s}{\Gamma \vdash_T^d s} \neg \neg \text{e} \quad \frac{\Gamma \vdash_T^d s : o \quad \Gamma, s \vdash_T^d t : o}{\Gamma \vdash_T^d (s \Rightarrow t) : o} \Rightarrow \text{type} \quad \frac{\Gamma \vdash_T^d s : o \quad \Gamma, s \vdash_T^d t}{\Gamma \vdash_T^d s \Rightarrow t} \Rightarrow \text{i} \\
\frac{\Gamma \vdash_T^d s \Rightarrow t \quad \Gamma \vdash_T^d s}{\Gamma \vdash_T^d t} \Rightarrow \text{e} \quad \frac{\Gamma, x : A \vdash_T^d s : o}{\Gamma \vdash_T^d \forall x : A. s : o} \forall \text{type} \quad \frac{\Gamma, x : A \vdash_T^d s}{\Gamma \vdash_T^d \forall x : A. s} \forall \text{i} \\
\frac{\Gamma \vdash_T^d \forall x : A. s \quad \Gamma \vdash_T^d t : A}{\Gamma \vdash_T^d s[x/t]} \forall \text{e} \quad \frac{\Gamma \vdash_T^d A \equiv A' \quad \Gamma, x : A \vdash_T^d s =_o s'}{\Gamma \vdash_T^d \forall x : A. s =_o \forall x : A'. s'} \text{cong}\forall \\
\frac{\Gamma \vdash_T^d s =_o s' \quad \Gamma \vdash_T^d s'}{\Gamma \vdash_T^d s} \text{cong}\perp \quad \frac{\Gamma \vdash_T^d s \perp \quad \Gamma \vdash_T^d s(\neg \perp)}{\Gamma \vdash_T^d \forall x : o. s x} \text{boolExt} \\
\frac{\Gamma \vdash_T^d s : o \quad \Gamma, x : A \vdash_T^d s \quad A \text{ simple type}}{\Gamma \vdash_T^d s} \text{nonempty}
\end{array}$$

Fig. 1. Natural Deduction Calculus for DHOL

information on concrete base type arguments is restored with the help of a partial equivalence relation (PER) A^* for each type A . A PER is a symmetric, transitive relation. The elements on which it is also reflexive are intended to be the members of the original dependent type, i.e., $\Gamma \vdash_T^d s : A$ if and only if $\overline{\Gamma} \vdash_{\overline{T}}^s A^* \overline{s} \overline{s}$.

Definition 1. *The translation from DHOL to HOL is given by the erasure function $\overline{}$ as well as A^* which computes the formula representing the corresponding PER of a type A . The functions are mutually defined by recursion on the grammar of DHOL. The erasure of a theory (context) is defined as the theory (context) which consists of its erased components.*

$$\begin{array}{ll}
\overline{o} = o & \overline{a t_1 \dots t_n} = a \\
\overline{\Pi x: A.B} = \overline{A} \rightarrow \overline{B} & \overline{\overline{x}} = x \\
\overline{\lambda x: A.s} = \lambda x: \overline{A}. \overline{s} & \overline{\overline{s t}} = \overline{s} \overline{t} \\
\overline{\perp} = \perp & \overline{\neg s} = \neg \overline{s} \\
\overline{s \Rightarrow t} = \overline{s} \Rightarrow \overline{t} & \overline{s =_A t} = A^* \overline{s} \overline{t} \\
\overline{\forall x: A.s} = \forall x: \overline{A}. A^* x x \Rightarrow \overline{s} & \overline{x: A} = x: \overline{A}, A^* x x
\end{array}$$

$$\begin{aligned}
\overline{a: \Pi x_1: A_1. \dots \Pi x_n: A_n. \text{tp}} &= a: \text{tp}, a^*: \overline{A_1} \rightarrow \dots \rightarrow \overline{A_n} \rightarrow a \rightarrow a \rightarrow o, a_{\text{per}} \\
& o^* s t = s =_o t \\
(a t_1 \dots t_n)^* s t &= a^* \overline{t_1} \dots \overline{t_n} s t \\
(\Pi x: A.B)^* s t &= \forall x, y: \overline{A}. A^* x y \Rightarrow B^* (s x) (t y)
\end{aligned}$$

Here, a_{per} is defined as follows:

$$a_{\text{per}} = \forall x_1: \overline{A_1}. \dots \forall x_n: \overline{A_n}. \forall u, v: a. a^* x_1 \dots x_n u v \Rightarrow u =_a v$$

Theorem 1 (Completeness [17]).

- if $\Gamma \vdash_T^d A: \text{tp}$ then $\overline{\Gamma} \vdash_{\overline{T}}^s \overline{A}: \text{tp}$ and A^* is a PER over \overline{A}
- if $\Gamma \vdash_T^d A \equiv B$ then $\overline{\Gamma} \vdash_{\overline{T}}^s \forall x, y: \overline{A}. A^* x y =_o B^* x y$
- if $\Gamma \vdash_T^d s: A$ then $\overline{\Gamma} \vdash_{\overline{T}}^s \overline{s}: \overline{A}$ and $\overline{\Gamma} \vdash_{\overline{T}}^s A^* \overline{s} \overline{s}$
- if $\Gamma \vdash_T^d s$ then $\overline{\Gamma} \vdash_{\overline{T}}^s \overline{s}$

Theorem 2 (Soundness [17]).

- if $\Gamma \vdash_T^d s: o$ and $\overline{\Gamma} \vdash_{\overline{T}}^s \overline{s}$ then $\Gamma \vdash_T^d s$
- if $\Gamma \vdash_T^d s: A$ and $\Gamma \vdash_T^d t: A$ and $\overline{\Gamma} \vdash_{\overline{T}}^s A^* \overline{s} \overline{t}$ then $\Gamma \vdash_T^d s =_A t$

Note that the erasure treats simple types and dependent types in the same way. In the following, we define a post-processing function Φ on top of the original erasure [17] which allows us to erase to simpler but equivalent formulas. The goal of Φ is to replace $A^* s t$ where A is a simple type by $s =_A t$. As a consequence, the guard $A^* x x$ in $\forall x: \overline{A}. s$ for simple types A can be removed. The following definition gives a presentation of Φ as a pattern rewrite system [13].

Definition 2. Given a HOL term s , we define $\Phi(s)$ to be the HOL term which results from applying the following pattern rewrite rules exhaustively to all sub-terms in a bottom-up fashion:

$$\begin{aligned} a^* F G &\rightarrow F =_a G \\ \forall x, y: A. (x =_A y) \Rightarrow (F x =_B G y) &\rightarrow F =_{A \rightarrow B} G \\ \forall x: A. (x =_A x) \Rightarrow F x &\rightarrow \forall x: A. F x \end{aligned}$$

Here, F, G are free variables for terms, a^* denotes the constant for the PER of a simple base type a and A, B are placeholders for simple types. Given a HOL theory T , there are finitely many instances for a^* but infinite choices for A and B , so the pattern rewrite system is infinite.

Lemma 1. Assume $\Gamma \vdash_T^d s: o$. $\Gamma \vdash_T^d s$ if and only if $\bar{\Gamma} \vdash_{\bar{T}}^s \Phi(\bar{s})$.

Proof. Since the erasure is sound and complete (Theorem 2 and Theorem 1), it suffices to show that $\bar{\Gamma} \vdash_{\bar{T}}^s \Phi(\bar{s})$ if and only if $\bar{\Gamma} \vdash_{\bar{T}}^s \bar{s}$. Consider the rules from Definition 2. $\Phi(\bar{s})$ is well-defined: Clearly, the rules terminate and confluence follows from the lack of critical pairs [13]. Hence, it is sufficient to prove $\bar{\Gamma} \vdash_{\bar{T}}^s l$ if and only if $\bar{\Gamma} \vdash_{\bar{T}}^s r$ for every rule in Definition 2. For the first rule, assume $\bar{\Gamma} \vdash_{\bar{T}}^s a^* F G$. Since $a_{\text{per}} \in \bar{T}$, we have $\bar{\Gamma} \vdash_{\bar{T}}^s F =_a G$. Now assume $\bar{\Gamma} \vdash_{\bar{T}}^s F =_a G$. Since F has type a , we obtain $\Gamma \vdash_T^d F =_a F$. Completeness of the erasure yields $\bar{\Gamma} \vdash_{\bar{T}}^s a^* F F$. Now, the assumption allows us to replace equals by equals, so we conclude $\bar{\Gamma} \vdash_{\bar{T}}^s a^* F G$. The desired result for the second rule follows from extensionality. Finally, the third rule is an easy logical simplification. \square

Given a theory T (context Γ) we write $\Phi(\bar{T})$ ($\Phi(\bar{\Gamma})$) to denote its erased version where formulas have been simplified with Φ .

Corollary 1. Assume $\Gamma \vdash_T^d s: o$. $\Gamma \vdash_T^d s$ if and only if $\Phi(\bar{\Gamma}) \vdash_{\Phi(\bar{T})}^s \Phi(\bar{s})$.

Example 2. Consider again the axiom recursively defining `app` from Example 1

$$\forall n, m: \text{nat}, z: \text{elem}, x: \text{lst } n, y: \text{lst } m.$$

$$\text{app } (s \ n) \ m \ (\text{cons } n \ z \ x) \ y =_{\text{lst } (s \ (\text{plus } n \ m))} \ \text{cons } (\text{plus } n \ m) \ z \ (\text{app } n \ m \ x \ y)$$

which we refer to as s_{app} . Its post-processed erasure $\Phi(\overline{s_{\text{app}}})$ is given by the following formula which is simpler than $\overline{s_{\text{app}}}$:

$$\begin{aligned} \forall n, m: \text{nat}, z: \text{elem}, x: \text{lst}. \text{lst}^* \ n \ x \ x \Rightarrow \forall y: \text{lst}. \text{lst}^* \ m \ y \ y \Rightarrow \text{lst}^* \ (s \ (\text{plus } n \ m)) \\ (\text{app } (s \ n) \ m \ (\text{cons } n \ z \ x) \ y) \ (\text{cons } (\text{plus } n \ m) \ z \ (\text{app } n \ m \ x \ y)) \end{aligned}$$

3 Tableau Calculus for DHOL

3.1 Rules

The tableau calculus from [1, 7] is the basis of Satallax [4] and its fork Lash [6]. We present an extension of this calculus from HOL to DHOL by extending

the rules to DHOL as well as providing tableau rules for the translation from DHOL to HOL. A *branch* is a 3-tuple (T, Γ, Γ') which is *well-formed* if $\vdash^d T \text{Thy}$, $\vdash_T^d \Gamma \text{Ctx}$ and $\vdash_{\Phi(\bar{T})}^s \Gamma' \text{Ctx}$. Intuitively, the theory contains the original problem and remains untouched while the contexts grow by the application of rules. Furthermore, DHOL and HOL are represented separately: For DHOL, the theory T and context Γ are used while HOL has a separate context Γ' with respect to the underlying theory $\Phi(\bar{T})$. In particular, each rule in Fig. 2 really stands for two rules: one that operates in DHOL and the original version that operates in HOL. Except for the erasure rules $\mathcal{T}_{\text{ER}_1}$ and $\mathcal{T}_{\text{ER}_2}$ which add formulas to the HOL context based on information from the DHOL theory and context, the rules always stay in DHOL or HOL, respectively. More formally, a *step* is an $n + 1$ -tuple $\langle (T, \Gamma, \Gamma'), (T, \Gamma_1, \Gamma'_1), \dots, (T, \Gamma_n, \Gamma'_n) \rangle$ of branches where $\perp \notin T, \Gamma, \Gamma'$ and either $\Gamma \subset \Gamma_i$ and $\Gamma' = \Gamma'_i$ for all $1 \leq i \leq n$ or $\Gamma = \Gamma_i$ and $\Gamma' \subset \Gamma'_i$ for all $1 \leq i \leq n$. Given a step $\langle A, A_1, \dots, A_n \rangle$, the branch A is called its *head* and each A_i is an *alternative*.

A *rule* is a set of steps defined by a schema. For example, the rule $\mathcal{T}_{\Rightarrow}$ from Fig. 2 indicates the set of steps $\langle (T, \Gamma, \Gamma'), (T, \Gamma_1, \Gamma'_1), (T, \Gamma_2, \Gamma'_2) \rangle$ where $\perp \notin T, \Gamma, \Gamma'$ and either $s \Rightarrow t \in T, \Gamma$ or $s \Rightarrow t \in \Phi(\bar{T}), \Gamma'$. In the former case, we have $\Gamma_1 = \Gamma, \neg s$ and $\Gamma_2 = \Gamma, t$ as well as $\Gamma' = \Gamma'_1 = \Gamma'_2$. The latter case is the same but with the primed and unprimed variants swapped.

In the original tableau calculus [1, 7], normalization is defined with respect to an axiomatized generic operator $[\cdot]$. As one would expect, one of these axioms states that the operator does not change the semantics of a given term. Since there is no formal definition of DHOL semantics yet, we simply use $[s]$ to denote the $\beta\eta$ -normal form of s which is in accordance with our implementation.

A rule *applies* to a branch A if some step in the rule has A as its head. A tableau calculus is a set of steps. Let \mathcal{T} be the tableau calculus defined by the rules in Fig. 2. The side condition of freshness in $\mathcal{T}_{\neg\forall}$ means that for a given step with head (T, Γ, Γ') there is no type A such that $y : A \in T, \Gamma$ or $y : A \in \Phi(\bar{T}), \Gamma'$ and we additionally require that there is no name x such that $\neg[s x] \in T, \Gamma$ or $\neg[s x] \in \Phi(\bar{T}), \Gamma'$. In practice, this means that to every formula, $\mathcal{T}_{\neg\forall}$ can be applied at most once. Furthermore, the side condition $t : A$ in the rule \mathcal{T}_{\forall} means that either $\Gamma \vdash_T^d t : A$ or $\Gamma' \vdash_{\Phi(\bar{T})}^s t : A$ depending on whether the premise is in T, Γ or $\Phi(\bar{T}), \Gamma'$. The side condition $\bar{s} : o$ in the rule $\mathcal{T}_{\text{ER}_1}$ means that $\Gamma' \vdash_{\Phi(\bar{T})}^s \bar{s} : o$. This is to prevent application of $\mathcal{T}_{\text{ER}_1}$ before the necessary type information is obtained by applying $\mathcal{T}_{\text{ER}_2}$.

The set of \mathcal{T} -*refutable* branches is defined inductively: If $\perp \in T, \Gamma, \Gamma'$, then (T, Γ, Γ') is refutable. If $\langle A, A_1, \dots, A_n \rangle$ is a step in \mathcal{T} and every alternative A_i is refutable, then A is refutable.

The rules in Fig. 2 strongly resemble the tableau calculus from [1]. In order to support DHOL, we replaced simple types by their dependent counterparts. To that end, we tried to remain as simple as possible by only allowing syntactically equivalent types in \mathcal{T}_{\forall} and \mathcal{T}_{CON} : Adding a statement like $A \equiv A'$ as a premise would change the tableau calculus as well as the automated proof search signif-

$$\begin{array}{c}
\mathcal{T}_{\neg} \frac{s, \neg s}{\perp} \quad \mathcal{T}_{\neq} \frac{s \neq_{at_1 \dots t_n} s}{\perp} \quad \mathcal{T}_{\neg\neg} \frac{\neg\neg s}{s} \quad \mathcal{T}_{\Rightarrow} \frac{s \Rightarrow t}{\neg s \mid t} \quad \mathcal{T}_{\neg\Rightarrow} \frac{\neg(s \Rightarrow t)}{s, \neg t} \\
\mathcal{T}_{\forall} \frac{\forall x: A.s}{[s[x/t]]} t: A \quad \mathcal{T}_{\neg\forall} \frac{\neg\forall x: A.s}{y: A, \neg[s[x/y]]} y \text{ fresh} \quad \mathcal{T}_{\text{BE}} \frac{s \neq_o t}{s, \neg t \mid \neg s, t} \\
\mathcal{T}_{\text{BQ}} \frac{s =_o t}{s, t \mid \neg s, \neg t} \quad \mathcal{T}_{\text{FE}} \frac{s \neq_{\Pi x: A.B} t}{\neg[\forall x. sx = tx]} x \notin \mathcal{V}_s \cup \mathcal{V}_t \quad \mathcal{T}_{\text{FQ}} \frac{s =_{\Pi x: A.B} t}{[\forall x. sx = tx]} x \notin \mathcal{V}_s \cup \mathcal{V}_t \\
\mathcal{T}_{\text{MAT}} \frac{x s_1 \dots s_n, \neg x t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n} n \geq 1 \quad \mathcal{T}_{\text{DEC}} \frac{x s_1 \dots s_n \neq_{au_1 \dots u_n} x t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n} n \geq 1 \\
\mathcal{T}_{\text{CON}} \frac{s =_{at_1 \dots t_n} t, u \neq_{at_1 \dots t_n} v}{s \neq u, t \neq u \mid s \neq v, t \neq v} \quad \mathcal{T}_{\text{ER}_1} \frac{s}{[\Phi(\bar{s})]} \bar{s}: o \quad \mathcal{T}_{\text{ER}_2} \frac{x: A}{x: \bar{A}, A^* x x}
\end{array}$$

Fig. 2. Tableau rules for DHOL

icantly, so these situations are handled by the erasure for which the additional rules $\mathcal{T}_{\text{ER}_1}$, $\mathcal{T}_{\text{ER}_2}$ are responsible.

It is known that the restriction of \mathcal{T} to HOL (without $\mathcal{T}_{\text{ER}_1}$ and $\mathcal{T}_{\text{ER}_2}$) is sound and complete with respect to Henkin semantics [1, 7]. Furthermore, due to Corollary 1, the rules $\mathcal{T}_{\text{ER}_1}$ and $\mathcal{T}_{\text{ER}_2}$ define a sound and complete translation from DHOL to HOL with respect to Rothgang et al.'s definition of provability in DHOL [17].

3.2 Soundness and Completeness

In general, a soundness result based on the refutability of a branch (T, Γ, Γ') is desirable. If there were a definition of semantics for DHOL which is a conservative extension of Henkin semantics, the proof could just refer to satisfiability of T, Γ, Γ' . Unfortunately, this is not the case. Note that an appropriate definition of semantics is out of the scope of this paper: In addition to its conception, we would have to prove soundness and completeness of \vdash^d on top of the corresponding proofs for our novel tableau calculus. Therefore, soundness and completeness of the tableau calculus will be established with respect to provability in DHOL or HOL. Unfortunately, this requirement complicates the proof tremendously as a refutation can contain a mixture of DHOL, erasure and HOL rules. Therefore, we have to consider both HOL and DHOL and need to establish a correspondence between Γ and Γ' which is difficult to put succinctly and seems to be impossible without further restricting the notion of a well-formed branch. Therefore, we prove soundness and completeness with respect to a notion of refutability which has three stages: At the beginning, only DHOL rules are applied, the second stage is solely for the erasure and in the last phase, only HOL rules are applied. Note that this notion of refutability includes the sound but incomplete strategy of only using native DHOL rules as well as the sound and complete strategy of exclusively working with the erasure.

Definition 3. A branch (T, Γ, Γ') is s-refutable if it is refutable with respect to the HOL rules.

Lemma 2. A well-formed branch (T, Γ, Γ') is s-refutable $\iff \Gamma' \vdash_{\Phi(\overline{T})}^s \perp$.

Proof. Immediate from soundness and completeness of the original HOL calculus as well as soundness and completeness of \vdash^s . \square

Definition 4. The set of e-refutable branches is inductively defined as follows: If (T, Γ, Γ') is s-refutable and $\Gamma' \subseteq \Phi(\overline{T})$, then it is e-refutable. If $\langle A, A_1 \rangle \in \mathcal{T}_{\text{ER}_1} \cup \mathcal{T}_{\text{ER}_2}$ and A_1 is e-refutable, then A is e-refutable.

Lemma 3. If (T, Γ, Γ') is well-formed and e-refutable then $\Phi(\overline{T}) \vdash_{\Phi(\overline{T})}^s \perp$.

Proof. Let (T, Γ, Γ') be well-formed and e-refutable. We proceed by induction on the definition of e-refutability. If (T, Γ, Γ') is s-refutable then $\Gamma' \vdash_{\Phi(\overline{T})}^s \perp$ by Lemma 2. Since $\Gamma' \subseteq \Phi(\overline{T})$ we also have $\Phi(\overline{T}) \vdash_{\Phi(\overline{T})}^s \perp$. For the induction step, let $\langle (T, \Gamma, \Gamma'), (T, \Gamma, \Gamma'_1) \rangle$ be a step with either $\mathcal{T}_{\text{ER}_1}$ or $\mathcal{T}_{\text{ER}_2}$ and assume that the branch (T, Γ, Γ'_1) is e-refutable. Since well-formedness of (T, Γ, Γ'_1) follows from the well-formedness of (T, Γ, Γ') , the induction hypothesis yields $\Phi(\overline{T}) \vdash_{\Phi(\overline{T})}^s \perp$ as desired. \square

Definition 5. The set of d-refutable branches is inductively defined as follows: If (T, Γ, \cdot) is e-refutable or $\perp \in T, \Gamma$, then it is d-refutable. If $\langle A, A_1, \dots, A_n \rangle \in \mathcal{T} \setminus (\mathcal{T}_{\text{ER}_1} \cup \mathcal{T}_{\text{ER}_2})$ and every alternative A_i is d-refutable, then A is d-refutable.

Next, we have to prove soundness of every DHOL rule. For most of the rules, this is rather straightforward. We show soundness of \mathcal{T}_{FE} , \mathcal{T}_{FQ} and \mathcal{T}_{DEC} as representative cases and start with an auxiliary lemma.

Lemma 4. Assume $\Gamma \vdash_T^d s : o$. We have $\Gamma \vdash_T^d s$ if and only if $\Gamma \vdash_T^d [s]$.

Proof. By the beta and eta rules, we have $\Gamma \vdash_T^d s =_o [s]$. Using $\text{cong}\vdash$ we obtain the desired result in both directions. \square

Lemma 5 (\mathcal{T}_{FE}). Let (T, Γ, Γ') be a well-formed branch. Choose x such that $x \notin \mathcal{V}s \cup \mathcal{V}t$ and assume $s \neq_{\Pi x: A.B} t \in T, \Gamma$. If $\Gamma, \neg[\forall x: A.sx = tx] \vdash_T^d \perp$ then $\Gamma \vdash_T^d \perp$.

Proof. From the assumptions and Lemma 4, we obtain $\Gamma \vdash_T^d s \neq_{\Pi x: A.B} t$ and $\Gamma \vdash_T^d \forall x: A.sx =_B tx$. Furthermore, an application of $\forall e$ yields $\Gamma, x: A \vdash_T^d sx =_B tx$. Using $\text{cong}\lambda$, we get $\Gamma \vdash_T^d (\lambda x: A.sx) =_{\Pi x: A.B} (\lambda x: A.tx)$. Hence, we can apply eta ($x \notin \mathcal{V}s \cup \mathcal{V}t$), sym and the admissible rule trans [18] which says that equality is transitive to get $\Gamma \vdash_T^d s =_{\Pi x: A.B} t$ and therefore $\Gamma \vdash_T^d \perp$. \square

Lemma 6 (\mathcal{T}_{FQ}). Let (T, Γ, Γ') be a well-formed branch. Assume $s =_{\Pi x: A.B} t \in T, \Gamma$ and $x \notin \mathcal{V}s \cup \mathcal{V}t$. If $\Gamma, [\forall x: A.sx = tx] \vdash_T^d \perp$ then $\Gamma \vdash_T^d \perp$.

Proof. From the assumptions, $\Gamma \vdash_T^d \neg[s] =_o [\neg s]$, $\text{cong}\vdash$ and Lemma 4, we obtain $\Gamma \vdash_T^d s =_{\Pi x: A.B} t$ and $\Gamma \vdash_T^d \neg\forall x: A.sx =_B tx$. Furthermore, we have $\Gamma, x: A \vdash_T^d sx =_B tx$ by refl and congApp . Hence, $\forall i$ yields $\Gamma \vdash_T^d \forall x: A.sx =_B tx$ and we conclude by an application of $\neg e$. \square

Lemma 7 (\mathcal{T}_{DEC}). *Let (T, Γ, Γ') be a well-formed branch. Assume*

$$x s_1 \dots s_n \neq_{au_1 \dots u_m} x t_1 \dots t_n \in T, \Gamma$$

and $\Gamma \vdash_T^d x: \Pi y_1: A_1 \dots \Pi y_n: A_n. a u'_1 \dots u'_m$ where $u_i = u'_i[y_1/s_1 \dots y_n/s_n]$ for $1 \leq i \leq m$. If $\Gamma, s_i \neq_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i \vdash_T^d \perp$ for all $1 \leq i \leq n$ then $\Gamma \vdash_T^d \perp$.

Proof. From the assumptions, we obtain $\Gamma \vdash_T^d s_i =_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i$ for all $1 \leq i \leq n$ and $\Gamma \vdash_T^d x =_{\Pi y_1: A_1 \dots \Pi y_n: A_n. au'_1 \dots u'_m} x$. Hence, n applications of the congruence rule for application yield $\Gamma \vdash_T^d x s_1 \dots s_n =_{au_1 \dots u_m} x t_1 \dots t_n$. Since we also have $\Gamma \vdash_T^d x s_1 \dots s_n \neq_{au_1 \dots u_m} x t_1 \dots t_n$, we obtain $\Gamma \vdash_T^d \perp$. \square

Now we are ready to prove the soundness result for \mathcal{T} .

Theorem 3. *If (T, Γ, \cdot) is well-formed and d-refutable then $\Gamma \vdash_T^d \perp$.*

Proof. Let (T, Γ, \cdot) be well-formed and d-refutable. We proceed by induction on the definition of d-refutability. If (T, Γ, \cdot) is e-refutable, the result follows from Lemma 3 together with Corollary 1. If $\perp \in T, \Gamma$ then clearly $\Gamma \vdash_T^d \perp$. For the inductive case, consider a step $\langle (T, \Gamma, \cdot), (T, \Gamma_1, \cdot), \dots, (T, \Gamma_n, \cdot) \rangle$ with some DHOL rule. Since (T, Γ, \cdot) is d-refutable, all alternatives must be d-refutable. If we manage to show well-formedness of every alternative, we can apply the induction hypothesis to obtain $\Gamma_i \vdash_T^d \perp$ for all $1 \leq i \leq n$. Then, we can conclude $\Gamma \vdash_T^d \perp$ by soundness of the DHOL rules. Hence, it remains to prove well-formedness of the alternatives. In most cases, this is straightforward. We only show one interesting case, namely \mathcal{T}_{DEC} .

Instead of proving $\Gamma, s_i \neq_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i \vdash_T^d \perp$ for all $1 \leq i \leq n$ we show that $\Gamma \vdash_T^d s_i =_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i$ for all $1 \leq i \leq n$. Since (T, Γ, \cdot) is a well-formed branch, both s_1 and t_1 have type A_1 . Hence, $(T, (\Gamma, s_1 \neq_{A_1} t_1), \cdot)$ is well-formed and our original induction hypothesis yields $\Gamma, s_1 \neq_{A_1} t_1 \vdash_T^d \perp$ from which we obtain $\Gamma \vdash_T^d s_1 =_{A_1} t_1$. Now let $i \leq n$ and assume we have $\Gamma \vdash_T^d s_j =_{A_j[x_1/s_1, \dots, x_{j-1}/s_{j-1}]} t_j$ for all $j < i$ (*). This is only possible if $\Gamma \vdash_T^d t_j: A_j[x_1/s_1, \dots, x_{j-1}/s_{j-1}]$ for all $j < i$. Since (T, Γ, \cdot) is a well-formed branch, it is clear that $\Gamma \vdash_T^d s_i: A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]$ and $\Gamma \vdash_T^d t_i: A_i[x_1/t_1, \dots, x_{i-1}/t_{i-1}]$. From (*), we obtain

$$\Gamma \vdash_T^d t_i: A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}],$$

so $(T, (\Gamma, s_i \neq_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i), \cdot)$ is well-formed. Hence, the original induction hypothesis yields $\Gamma \vdash_T^d s_i =_{A_i[x_1/s_1, \dots, x_{i-1}/s_{i-1}]} t_i$ as desired. \square

In the previous proof, we can see that for \mathcal{T}_{DEC} , well-formedness of an alternative depends on refutability of all branches to the left. Note that the same holds

for \mathcal{T}_{MAT} and $\mathcal{T}_{\Rightarrow}$. This is a distinguishing feature of DHOL as in tableaux, branches are usually considered to be independent.

Finally, completeness is immediate from the completeness of the HOL tableau calculus and the erasure:

Theorem 4. *If $\Gamma \vdash_T^d \perp$ then (T, Γ, \cdot) is d-refutable.*

Proof. Let $\Gamma \vdash_T^d \perp$. Using Corollary 1 and Lemma 2 we conclude s-refutability of $(T, \Gamma, \Phi(\overline{\Gamma}))$. By definition, $(T, \Gamma, \Phi(\overline{\Gamma}))$ is also e-refutable. Furthermore, by inspecting $\mathcal{T}_{\text{ER}_1}$ and $\mathcal{T}_{\text{ER}_2}$ we conclude that (T, Γ, \cdot) is also e-refutable and therefore d-refutable. \square

4 Implementation

We implemented the tableau calculus for DHOL as an extension of Lash [6] which is a fork of Satallax, a successful automated theorem prover for HOL [4]. By providing an efficient C implementation of terms with perfect sharing as well as other important data structures and operations, Lash outperforms Satallax when it comes to the basic ground tableau calculus which both of them implement. However, Lash removes a lot of the additional features beyond the basic calculus that was implemented in Satallax. Nevertheless, this was actually beneficial for our purpose as we could concentrate on adapting the core part. Note that Lash and Satallax do not just implement the underlying ground tableau calculus but make heavy use of SAT-solving and a highly customizable priority queue to guide the proof search [4, 5].

For the extension of Lash to DHOL, the data structure for terms had to be changed to support dependent function types as well as quantifiers and lambda abstractions with dependent types. Of course, it would be possible to represent everything in the language of DHOL but the formulation of DHOL suggests that the prover should do as much as possible in the HOL fragment and only use “proper” DHOL when it is really necessary. With this in mind, the parser first always tries to produce simply-typed terms and only resorts to dependent types when it is unavoidable. Therefore, the input problem often looks like a mixture of HOL and DHOL even though everything is included in DHOL. A nice side effect of this design decision is that our extension of Lash works exactly like the original version on the HOL fragment except for the fact that it is expected to be slower due to the numerous case distinctions between simple types and dependent types which are needed in this setting.

Although DHOL is not officially part of TPTP THF, it can be expressed due to the existence of the $!>$ -symbol which is used for polymorphism. Hence, a type $\Pi x: A. B$ is represented as $!>[X:A]:B$. For simplicity and efficiency reasons, we did not implement dependent types by distinguishing base types from their term arguments but represent the whole dependent type as a term. When parsing a base type a , Lash automatically creates an eponymous constant of type tp to be used in dependent types as well as a simple base type a_0 for the erasure and a constant a^* for its PER. The flags `DHOL_RULES_ONLY` and

DHOL_ERASURE_ONLY control the availability of the erasure as well as the native DHOL rules, respectively. Note that the implementation is not restricted to d-refutability but allows for arbitrary refutations. In the standard flag setting, however, only the native DHOL rules are used. Clearly, this constitutes a sound strategy. It is incomplete since the confrontation rule only considers equations with syntactically equivalent types. We have more to say about this in Sect. 4.2.

4.1 Type Checking

By default, problems are only type-checked with respect to their simply-typed skeleton. If the option `exactdholytypecheck` is set, type constraints stemming from the term arguments of dependent base types are generated and added to the conjecture. The option `typecheckonly` discards the original conjecture, so Lash just tries to prove the type constraints. Since performing the type check involves proper theorem proving, we added the new SZS ontology statuses `TypeCheck` and `InexactTypecheck` to the standardized output of Lash. Here, the former one means that a problem type checks while the latter one just states that it type checks with respect to the simply-typed skeleton.

For the generation of type constraints, each formula of the problem is traversed like in normal type checking. In addition, every time a type condition $a t_1 \dots t_n \equiv a s_1 \dots s_n$ comes up and there is some i such that s_i and t_i are not syntactically equivalent, a constraint stating that $s_i = t_i$ is provable is added to the set of type constraints. Note that it does not always suffice to just add $s_i = t_i$ as this equation may contain bound variables or only hold in the context in which the constraint appears. To that end, we keep relevant information about the term context when generating these constraints. Whenever a forall quantifier or lambda abstraction comes up, it is translated to a corresponding forall quantifier in the context since we want the constraint to hold in any case. While details like applications can be ignored, it is important to keep left-hand sides of implications in the context as it may be crucial for the constraint to be met. In general, any axiom may contribute to the typechecking proof.

Example 3. The conjecture

$$\forall n: \text{nat}, x: \text{lst } n. n =_{\text{nat}} 0 \Rightarrow \text{app } n n x x = x$$

is well-typed if the type constraint

$$\forall n: \text{nat}, x: \text{lst } n. n =_{\text{nat}} 0 \Rightarrow \text{plus } n n =_{\text{nat}} n$$

is provable. Lash can generate this constraint and finds a proof quickly using the axiom $\forall n: \text{nat}. \text{plus } 0 n =_{\text{nat}} n$.

Since conjunctions and disjunctions are internally translated to implications, it is important to note that we process formulas from left to right, i.e. for $x: \text{lst } n$ and $y: \text{lst } m$, the proposition $m \neq n \vee x = y$ type checks because we can assume $m = n$ to process $x = y$. Consequently, $x = y \vee m \neq n$ does not type check.

As formulas are usually read from left to right, this is a natural adaption of short-circuit evaluation in programming languages. Furthermore, it is in accordance with the presentation of Rothgang et al. [17] as well as the corresponding implementation in PVS [19]. As a matter of fact, PVS handles its undecidable type checking problem in essentially the same way as our new version of Lash by generating so called *type correctness conditions* (TCCs).

4.2 Implementation of the Rules

Given the appropriate infrastructure for dependent types, the implementation of most rules in Fig. 2 is a straightforward extension of the original HOL implementation. For \mathcal{T}_\forall , the side condition $\Gamma \vdash_T^d t : A$ is undecidable in general. It has been chosen to provide a simple characterization of the tableau calculus. Furthermore, it emphasizes that we do not instantiate with terms whose type does not literally match with the type of the quantified variable. In the implementation, we keep a pool of possible instantiations for types A which occur in the problem. The pool gets populated by terms of which we know that they have a given type because this information was available during parsing or proof search. Hence, we only instantiate with terms t for which we already know that $\Gamma \vdash_T^d t : A$ holds.

Given an equation $s =_A t$, there are many candidate representations of A modulo type equality. When we build an equation in the implementation, we usually use the type of the left-hand side. Since all native DHOL rules of the tableau calculus enforce syntactically equivalent types, the ambiguity with respect to the type of an equation leads to problems. For example, consider a situation where $\Gamma \vdash_T^d s : A$, $\Gamma \vdash_T^d t : B$ and $\Gamma \vdash_T^d s =_A t$ which implies $\Gamma \vdash_T^d A \equiv B$. During proof search, it could be that $\Gamma \vdash_T^d t \neq s$ is established. Clearly, this is a contradiction which leads to a refutation, but usually the inequality annotated with the type B which makes the refutation inaccessible for our native DHOL rules. Therefore, we implemented rules along the lines of

$$\mathcal{T}_{\text{SYMCAST}_1} \frac{s =_A t}{t =_B s} t : B \quad \mathcal{T}_{\text{SYMCAST}_2} \frac{s \neq_A t}{t \neq_B s} t : B$$

which do not only apply symmetry but also change the type of the equality in a sound way. Like in \mathcal{T}_\forall , the side condition should be read as $\Gamma \vdash_T^d t : B$ which makes it undecidable. However, in practice, we can compute a representative of the type of t given the available type information. While experimenting with the new DHOL version of Lash, the implementation of these rules proved to be very beneficial for refutations which only work with the DHOL rules. For the future, it is important to note that $\mathcal{T}_{\text{SYMCAST}_1}$ and $\mathcal{T}_{\text{SYMCAST}_2}$ are not sound for the extension of DHOL to predicate subtypes as $\Gamma \vdash_T^d s =_A t$ and $\Gamma \vdash_T^d t : B$ do not imply $\Gamma \vdash_T^d A \equiv B$ anymore.

4.3 Generating Instantiations

Since Lash implements a ground tableau calculus, it does not support higher-order unification. Therefore, the generation of suitable instantiations is a major

issue. In the case of DHOL, it is actually beneficial that Lash already implements other means of generating instantiations since the availability of unification for DHOL is questionable: There exist unification procedures for dependent type theories (see for example [11]) but for DHOL such a procedure would also have to address the undecidable type equality problem.

For simple base types, it suffices to consider so-called *discriminating* terms to remain complete [1]. A term s of simple base type a is discriminating in a branch A if $s \neq_a t \in A$ or $t \neq_a s \in A$ for some term t . For function terms, completeness is guaranteed by enumerating all possible terms of a given type. Of course, this is highly impractical, and there is the important flag `INITIAL_SUBTERMS_AS_INSTANTIATIONS` which adds all subterms of the initial problem as instantiations. This heuristic works very well in many cases.

For dependent types, we do not check for type equality when instantiating quantifiers but only use instantiations with the exact same type (c.f. \mathcal{T}_\forall in Fig. 2) and let the erasure handle the remaining cases.

An interesting feature of this new version of Lash is the possibility to automatically generate instantiations for induction axioms. Given the constraints of the original implementation, the easiest way to sneak a term into the pool of instantiations is to include it into an easily provable lemma and then use the flag `INITIAL_SUBTERMS_AS_INSTANTIATIONS`. However, this adds unnecessary proof obligations, so we modified the implementation such that initial subterms as instantiations also include lambda-abstractions corresponding to forall quantifiers.

Example 4. Consider the induction axiom for lists:

$$\begin{aligned} \forall p: (\Pi n: \text{nat}. \text{lst } n \rightarrow o). p \ 0 \ \text{nil} \\ \Rightarrow (\forall n: \text{nat}, x: \text{elem}, y: \text{lst } n. p \ n \ y \Rightarrow p \ (s \ n) \ (\text{cons } n \ x \ y)) \\ \Rightarrow (\forall n: \text{nat}, x: \text{lst } n. p \ n \ x) \end{aligned}$$

Even though it works for arbitrary predicates p , it is very hard for an ATP system to guess the correct instance for a given problem without unification in general. However, given the conjecture $\forall n: \text{nat}, x: \text{lst } n. \text{app } n \ 0 \ x \ \text{nil} =_{\text{lst } n} x$ we can easily read off the correct instantiation for p where \forall is replaced by λ .

5 Case Study: List Reversal Is an Involution

Consider the following equational definition of the list reversal function `rev`:

$$\begin{aligned} \text{rev } 0 \ \text{nil} &=_{\text{lst } 0} \ \text{nil} \\ \forall n: \text{nat}, x: \text{elem}, y: \text{lst } n. \\ \text{rev } (s \ n) \ (\text{cons } n \ x \ y) &=_{\text{lst } (s \ n)} \ \text{app } n \ (s \ 0) \ (\text{rev } n \ y) \ (\text{cons } 0 \ x \ \text{nil}) \end{aligned}$$

The conjecture

$$\forall n: \text{nat}, x: \text{lst } n. \text{rev } n \ (\text{rev } n \ x) =_{\text{lst } n} x \quad (\text{rev-invol})$$

Table 1. Amount of problem files per (intermediate) goal

Goal	Number of Problem Files
<code>app-nil</code>	4
<code>app-assoc</code>	8
<code>app-assoc-m1</code>	5
<code>rev-invol-lem</code>	12
<code>rev-invol</code>	5

is very easy to state, but turns out to be hard to prove automatically. The proof is based on the equational definitions of `plus` and `app` given in Example 1 as well as several induction proofs on lists using the axiom from Example 4. In particular, some intermediate goals are needed to succeed:

$$\forall n: \text{nat}, x: \text{lst } n. \text{app } n \ 0 \ x \ \text{nil} =_{\text{lst } n} x \quad (\text{app-nil})$$

$$\begin{aligned} \forall n_1: \text{nat}, x_1: \text{lst } n_1, n_2: \text{nat}, x_2: \text{lst } n_2, n_3: \text{nat}, x_3: \text{lst } n_3. \quad & (\text{app-assoc}) \\ \text{app } n_1 \ (\text{plus } n_2 \ n_3) \ x_1 \ (\text{app } n_2 \ n_3 \ x_2 \ x_3) & \\ = \text{app } (\text{plus } n_1 \ n_2) \ n_3 \ (\text{app } n_1 \ n_2 \ x_1 \ x_2) \ x_3 & \end{aligned}$$

$$\begin{aligned} \forall n: \text{nat}, x: \text{lst } n, y: \text{elem}, m: \text{nat}, z: \text{lst } m. \quad & (\text{app-assoc-m1}) \\ \text{app } (\text{plus } n \ (\text{s } 0)) \ m \ (\text{app } n \ (\text{s } 0) \ x \ (\text{cons } 0 \ y \ \text{nil})) \ z = \text{app } n \ (\text{s } m) \ x \ (\text{cons } m \ y \ z) & \end{aligned}$$

$$\begin{aligned} \forall n: \text{nat}, x: \text{lst } n, m: \text{nat}, y: \text{lst } m. \quad & (\text{rev-invol-lem}) \\ \text{rev } (\text{plus } n \ m) \ (\text{app } n \ m \ (\text{rev } n \ x) \ y) = \text{app } m \ n \ (\text{rev } m \ y) \ x & \end{aligned}$$

Note that for polymorphic lists, this is a standard example of an induction proof with lemmas (see e.g. [15, Section 2.2]). In the dependently-typed case, however, many intermediate equations would be ill-typed in interactive theorem provers like Coq or Lean. In order to succeed in automatically proving these problems, we had to break them down into separate problems for the instantiation of the induction axiom, the base case and the step case of the induction proofs. Often, we further needed to organize these subproblems in manageable steps. Overall, we created 34 TPTP problem files which are distributed over the intermediate goals as shown in Table 1. Note that already type checking these intermediate problems is not trivial: All type constraints are arithmetic equations, and given the Peano axioms, many of them need to be proven by induction themselves. Since we are mainly interested in the dependently-typed part, we added the needed arithmetical facts as axioms. Overall, the problem files have up to 18 axioms including the Peano axioms, selected arithmetical results, the defining equations of `plus`, `app` and `rev` as well as the list induction axiom. We left out unnecessary axioms in many problem files to make the proof search feasible.

With our new modes for DHOL which solely work with the native DHOL rules, Lash can type check and prove all problems easily. If we turn off the native DHOL rules and only work with the erasure using the otherwise same modes with a 60 s timeout, Lash can still typecheck all problems but it only manages to prove 7 out of 34 problems. In order to further evaluate the effectiveness of our new implementation, we translated all problems from DHOL to HOL using the *Logic Embedding Tool*¹, which performs the erasure from [17]. We then tested 16 other HOL provers available on *SystemOnTPTP*² on the translated problems with a 60 s timeout (without type checking). We found that 5 of the 34 problems could only be solved by the DHOL version of Lash, including one problem where it only needs 5 inference steps. Detailed results as well as means to reproduce them are available on Lash’s website³ together with its source code.

6 Conclusion

Starting from the erasure from DHOL to HOL by Rothgang et al. [17], we developed a sound and complete tableau calculus for DHOL which we implemented in Lash. To the best of our knowledge, this makes it the first standalone automated theorem prover for DHOL. According to the experimental results, configurations where the erasure is performed as a preprocessing step for a HOL theorem prover can be outperformed by our new prover by solely using the native DHOL rules. We hope that this development will raise further interest in DHOL. Possible further work includes theoretical investigations such as the incorporation of choice operators into the erasure as well as a definition of the semantics of DHOL. Furthermore, it is desirable to officially define the TPTP syntax for DHOL which then opens the possibility of establishing a problem data set on which current and future tools can be compared. Finally, we would like to extend Lash to support predicate subtypes. Rothgang et al. already incorporated this into the erasure but there is no corresponding syntactic support in TPTP yet. In particular, this would get us much closer to powerful automation support for systems like PVS.

Acknowledgments. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This work has also received funding from the European Union’s Horizon Europe research and innovation programme under grant agreement no. 101070254 CORESENSE as well as the ERC PoC grant no. 101156734 *FormalWeb3*. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Horizon Europe programme. Neither the European Union nor the granting authority can be held responsible for them.

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

¹ <https://github.com/leoprover/logic-embedding>.

² <https://tptp.org/cgi-bin/SystemOnTPTP>.

³ <http://cl-informatik.uibk.ac.at/software/lash-dhol/>.

References

1. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. *J. Autom. Reason.* **47**, 451–479 (2011). <https://doi.org/10.1007/s10817-011-9233-2>
2. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for higher-order logic. *J. Autom. Reason.* **67**, 10 (2023). <https://doi.org/10.1007/s10817-022-09649-9>
3. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_6
4. Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) *IJCAR 2012*. LNCS (LNAI), vol. 7364, pp. 111–117. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_11
5. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. *J. Autom. Reason.* **51**, 57–77 (2013). <https://doi.org/10.1007/s10817-013-9283-8>
6. Brown, C.E., Kaliszyk, C.: Lash 1.0 (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) *IJCAR 2022*. LNAI, vol. 13385, pp. 350–358. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_21
7. Brown, C.E., Smolka, G.: Analytic tableaux for simple type theory and its first-order fragment. *Log. Methods Comput. Sci.* **6**(2), 1–33 (2010). [https://doi.org/10.2168/LMCS-6\(2:3\)2010](https://doi.org/10.2168/LMCS-6(2:3)2010)
8. Castellan, S., Clairambault, P., Dybjer, P.: Undecidability of equality in the free locally cartesian closed category (extended version). *Log. Methods Comput. Sci.* **13**(4), 1–38 (2017). [https://doi.org/10.23638/LMCS-13\(4:22\)2017](https://doi.org/10.23638/LMCS-13(4:22)2017)
9. Church, A.: A formulation of the simple theory of types. *J. Symb. Log.* **5**(2), 56–68 (1940). <https://doi.org/10.2307/2266170>
10. Coq Development Team: *The Coq Reference Manual*, Release 8.18.0 (2023)
11. Elliott, C.M.: Higher-order unification with dependent function types. In: Dershowitz, N. (ed.) *RTA 1989*. LNCS, vol. 355, pp. 121–136. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51081-8_104
12. Martin-Löf, P.: Constructive mathematics and computer programming. *Philos. Trans. Royal Soc. Lond. A Math. Phys. Sci.* **312**, 501–518 (1984). <https://doi.org/10.1098/rsta.1984.0073>
13. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. *Theoret. Comput. Sci.* **192**(1), 3–29 (1998). [https://doi.org/10.1016/S0304-3975\(97\)00143-6](https://doi.org/10.1016/S0304-3975(97)00143-6)
14. Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) *CADE 2021*. LNCS (LNAI), vol. 12699, pp. 625–635. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_37
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
16. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: *CADE 1999*. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
17. Rothgang, C., Rabe, F., Benz Müller, C.: Theorem proving in dependently-typed higher-order logic. In: Pientka, B., Tinelli, C. (eds.) *CADE 2023*. LNAI, vol. 14132, pp. 438–455. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_25

18. Rothgang, C., Rabe, F., Benzmüller, C.: Theorem proving in dependently-typed higher-order logic – extended preprint (2023). <https://doi.org/10.48550/arXiv.2305.15382>
19. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.* **24**(9), 709–720 (1998). <https://doi.org/10.1109/32.713327>
20. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. *J. Autom. Reason.* **65**, 775–807 (2021). <https://doi.org/10.1007/s10817-021-09588-x>
21. Swamy, N., et al.: Dependent types and multi-monadic effects in F^* . In: Bodik, R., Majumdar, R. (eds.) *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 256–270 (2016). <https://doi.org/10.1145/2837614.2837655>

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

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





The Naproche-ZF Theorem Prover (Short Paper)

Adrian De Lon^(✉) 

University of Bonn, Bonn, Germany
adelon@uni-bonn.de

Abstract. Naproche-ZF is a new experimental open-source *natural* theorem prover based on set theory; formalizations in Naproche-ZF are written in a controlled natural language embedded into L^AT_EX and proof gaps are filled in with automated theorem provers. Naproche-ZF aims to scale natural theorem proving beyond chapter-sized formalizations. In contrast to the Naproche system, the new system uses an extensible grammar-based approach, has more efficient proof automation, and enables larger interconnected formalizations based on a standard library.

1 Introduction

Despite significant progress in theorem provers and successful formalizations of research-level mathematics [17], theorem provers have not yet enjoyed broad adoption among mathematicians [12, 29, 40]. A common criticism levelled against theorem provers by mathematicians is that formalizations are hard to write *and* read: they are written in unfamiliar languages, contain clutter that is irrelevant to the core mathematical ideas, may require knowledge of various specialized proof tactics, and are simply longer overall (see *de Bruijn factor* [39]).

Natural theorem provers are a direct answer to this critique; they aim to check texts *as written* by mathematicians: in natural language and with many proof gaps. Similar ambitions can already be found in the work of pioneers in theorem proving, such as P. Abrahams’s 1960s Proofchecker [1], which was intended to check the reasoning of textbooks as-is. Some theorem provers are *partly natural*, such as the influential Mizar system [16] which uses a quasi-natural input language and allows *obvious inferences* [34] as proof gaps. There are significant challenges to the natural approach, both in the processing of natural language and in the high degree of proof automation required to fill proof gaps. However, advances in automated theorem proving and computer hardware have made this approach more feasible.

Naproche [9, 26] is a natural theorem prover based on A. Paskevich’s implementation of SAD [27, 38], extending it with, e.g., set-theoretic primitives, more efficient checking, and an integrated development environment. Students have completed formalizations in Naproche in various areas of mathematics, such as analysis, axiomatic set theory, representation theory, and combinatorics. However, typical formalizations in Naproche use ad hoc axiomatic preliminaries and

struggle to scale beyond chapter-length. Medium-sized formalizations of around 3000 lines can take half an hour to check and proving new theorems becomes increasingly difficult.

Naproche-ZF¹ is a reimplementation of key ideas in Naproche with larger formalizations in mind. We shall compare the two systems throughout this paper. Naproche-ZF is developed in tandem with a growing modular standard library (see Footnote 1) containing formalizations of foundational material on sets, relations, functions, orders, ordinals, algebraic structures, topological spaces, and more.

2 Controlled Natural Language

The input language of Naproche-ZF is a new controlled natural language (CNL): it is a carefully chosen and formally specified subset of mathematical English that is embedded into \LaTeX for mathematical notation and document structuring. Most mathematicians are familiar with \LaTeX , which makes the language easier to learn. Ideally formalizing in a CNL should feel like writing with a strict style guide. The following example shows a theorem formalized in Naproche-ZF, first the \LaTeX source and then the rendering after typesetting.

```
\begin{theorem}[Burali-Forti antimony]\label{burali_forti}
  There exists no set  $\Omega$  such that for all  $\alpha$ 
  we have  $\alpha \in \Omega$  iff  $\alpha$  is an ordinal.
\end{theorem}
\begin{proof}
  Suppose not. Consider  $\Omega$  such that for all  $\alpha$  we have
   $\alpha \in \Omega$  iff  $\alpha$  is an ordinal.
  For all  $x, y$  such that  $x \in y \in \Omega$  we have  $x \in \Omega$ .
  So  $\Omega$  is  $\in$ -transitive. Thus  $\Omega$  is an ordinal.
  Hence  $\Omega \in \Omega$ . Contradiction.
\end{proof}
```

Theorem (Burali-Forti antimony). There exists no set Ω such that for all α we have $\alpha \in \Omega$ iff α is an ordinal.

Proof. Suppose not. Consider Ω such that for all α we have $\alpha \in \Omega$ iff α is an ordinal. For all x, y such that $x \in y \in \Omega$ we have $x \in \Omega$. So Ω is \in -transitive. Thus Ω is an ordinal. Hence $\Omega \in \Omega$. Contradiction. \square

Naproche-ZF treats everything outside of fixed *formal environments* such as **definition**, **theorem**, and **proof**, as comments. This facilitates writing *literate formalizations* that mix informal commentary and formal mathematics in the same document (e.g. [8], cf. literate programming [21]). Other theorem provers also support literate formalization or advanced typesetting; examples include Literate Agda, Isabelle’s documents, and a Mizar-to- \LaTeX translator [2]. An advantage of CNLs in literate formalization is that it is rarely necessary to restate theorems and proofs steps, since the formal statement is already readable.

¹ Available at <https://adelon.net/naproche-zf> under an open-source license together with its standard library.

Parsing. Natural mathematical language is *dynamic*: definitions introduce new lexical items, which can be symbols, words, or phrases. Dynamism complicates the processing of mathematical language. Naproche and Naproche-ZF take different approaches to parsing their input languages and modelling dynamism.

Naproche’s parser is defined with monadic parser combinators [18]. It statefully modifies itself as it encounters definitions and translates to an internal formula representation on the fly. Such tight coupling makes it harder to extend its CNL. There are also cases where parsing takes exponential time.

Naproche-ZF splits this process into phases. First it finds all lexical items using a scanner written with applicative regex combinators [7]. For nouns and verbs it then guesses the plural forms with basic *smart paradigms* [10] in the sense of GF [31]. The resulting lexicon becomes a parameter for the grammar of the CNL. Using the Earley [13] Haskell library, the CNL is specified as a context-free grammar in an embedded domain-specific language and the derived Earley parser [11] parses in cubic time in the worst-case or in quadratic time if the grammar is unambiguous. This grammar-oriented approach makes the initial design and future extensions of a CNL easier compared to parser combinators: new rules can be stated declaratively and there is no need to worry about exponential parsing times or eliminating left-recursion in the grammar.

Accuracy. Naproche supports a plain text dialect [28] and a L^AT_EX dialect. Naproche-ZF drops support for the plain text format and uses L^AT_EX markup in its CNL to avoid ambiguities. For instance, “a” can be ambiguous in Naproche (variable vs. determiner), but is clarified by distinguishing “a” (“ $\$a\$$ ”) from “a” (“a”). Such distinctions and avoidance of the backtracking behaviour of combinator parsers significantly improve error specificity and locality. For instance, Naproche often mistakes an unexpected word as a new variable name and will usually offer nonspecific and mislocated error messages along the lines of an “unexpected ”. ”, whereas Naproche-ZF can reliably offer a list of valid tokens at the location of the error. Naproche-ZF uses a stateful tokenizer to handle nested math and text modes, to support, e.g., “`\text{...}`” within set comprehensions.

Naproche accommodates grammatical number via a synonym instruction. For example, one uses “[`synonym number/-s`]” to identify “`natural number`” and “`natural numbers`”. Thus Naproche accepts ungrammatical sentences such as “`x,y is natural numbers`”. Naproche-ZF guesses plural forms via smart paradigms and requires number agreement. Overall, the grammar of Naproche-ZF is stricter with the aim of avoiding ambiguity, ungrammaticality, as well as pitfalls observed in formalizations written by students, where statements had unintuitive meanings in Naproche.

Naproche-ZF supports various idioms. For example, binary relations can be chained and multiple terms can be related to each other (“ $a, b < c < d$ ”), they can appear in bounded quantifiers (“For all $x \in X$...”), and sets can be used as binary relations (“ $x R y$ ”). Naproche-ZF also models some *pragmatic phenomena* [32, 35]: for example, an existential claim in a proof implicitly introduces a local constant, the same way that the “Consider ...” step does.

3 Semantics and Proof Checking

Translation. Naproche-ZF translates from its CNL to a set theory in higher-order logic (HOL) with Henkin semantics [5], using generalized de Bruijn indices [19, 20] to handle quantifiers and other binders. However, the system emphasizes reasoning within the first-order fragment where possible to use the strength of mature first-order automated theorem provers (ATPs).

Adjectives, verbs, and nouns are translated as predicates. Bounding phrases in quantifications are translated to type guards. For instance, “Every natural number is an ordinal” is translated to “ $\forall n. \text{natural_number}(n) \rightarrow \text{ordinal}(n)$ ”.

Proof automation is currently first-order only, using strong first-order ATPs such as E [36] and Vampire [22, 33]. Every nontrivial proof step or intermediate claim leads to a proof task that is exported to an ATP. By default an ATP is given 10s per tasks, but most tasks can be solved within fractions of a second. Naproche-ZF will also integrate with ATPs supporting HOL via TPTP THF0 [4]. For an impression of how ATPs are used, consider the step “Then $B \in 2^A$ ” in the following formalization of Cantor’s theorem.

Theorem (Cantor). There exists no surjection from A to 2^A .

Proof. Suppose not. Take a surjection f from A to 2^A . Let $B = \{a \in A \mid a \notin f(a)\}$.

Then $B \in 2^A$. There exists $a' \in A$ such that $f(a') = B$ by the definition of surjectivity. Now $a' \in B$ iff $a' \notin f(a') = B$. Contradiction.

This step leads to a proof task in which all preceding first-order definitions and theorems, as well as all local assumptions, definitions, and claims can be used as premises. Here the exported TPTP [37] problem contains a few hundred premises, shown below with the conjecture and recent theorems at the top, along with local premises at the bottom. Note that Naproche-ZF transformed the local definition of B via set comprehension into the first-order premise `cantor1` by an automatic application of the axiom of separation.

```
fof(cantor,conjecture,in(fb,pow(fA))).
[... ]
fof(powerset_intro,axiom,[XA,XB]:(subseteq(XA,XB)=>in(XA,pow(XB)))).
[... ]
fof(subseteq,axiom,[XA,XB]:(subseteq(XA,XB)<=>![Xa]:(in(Xa,XA)=>in(Xa,XB)))).
[... ]
fof(cantor1,axiom,[Xa]:(in(Xa,fb)<=>(in(Xa,fA)&~in(Xa,apply(ff,Xa)))).
fof(cantor2,axiom,in(ff,surj(fA,pow(fA)))).
fof(cantor3,axiom,~?[X5]:in(X5,surj(fA,pow(fA)))).
```

Sets. Naproche-ZF’s built-in constructs are geared towards higher-order set theories [5, 15] extending Zermelo–Fraenkel (ZF) set theory. ZF with the axiom of choice (ZFC) is the de facto foundation of informal mathematics, but additional axioms such as the universe axiom of Tarski–Grothendieck set theory (TG) can be convenient for, e.g., category theory. Variants of TG are used by Mizar, Egale, and Megalodon [6]. Second-order axioms of ZF have corresponding built-in syntax or proof steps in Naproche-ZF that make it possible to use them

with first-order proof automation. As we have seen in Cantor’s theorem above, set comprehensions are automatically eliminated in some situations using the axioms of separation or n -ary replacement. Naproche-ZF has a built-in proof method for \in -induction and will also feature a mechanism for defining one’s own induction principles (proved as higher-order theorems). There is potential for interoperability or integration with systems based on set theory, such as Mizar, Isabelle/ZF, and Megalodon. Naproche-ZF has an experimental export feature that translates theorem statements to Megalodon.

Structures. Naproche has no dedicated features for mathematical structures, which means that users have to set up structures themselves. Dealing with notation becomes cumbersome, as you have to explicitly annotate which structure an operation belongs to. In Naproche-ZF one can define structures directly. They are encoded as record datatypes in set theory, with structure operations acting as field projections. The noun phrase of a structure is translated as a predicate and structure axioms are translated to first-order introduction and elimination rules. This first-order encoding enables structures subtyping and multiple inheritance. In the example below, topological spaces inherit from the built-in *onesorted structure*, which has only a projection “ $|-$ ” to the carrier set as an operation and has no axioms.

Definition. A topological space X is a onesorted structure equipped with \mathcal{O}_X s.t.

1. \mathcal{O}_X is a family of subsets of $|X|$.
2. $\emptyset, |X| \in \mathcal{O}_X$.
3. For all $A, B \in \mathcal{O}_X$ we have $A \cap B \in \mathcal{O}_X$.
4. For all $F \subseteq \mathcal{O}_X$ we have $\bigcup F \in \mathcal{O}_X$.

Structure operations are typeset using L^AT_EX macros that take the structure as an optional argument. In theorems and proofs the structure argument may be omitted when a suitable structure was *instantiated* beforehand. For example, if a theorem statement has a premise of the form “Let X be a topological space”, one can subsequently write \mathcal{O} ($\$ \backslash \text{opens} \$$) for \mathcal{O}_X ($\$ \backslash \text{opens} [X] \$$). When multiple structures with the same operation are instantiated, the last instantiation shadows the previous ones.

Imports. The import mechanism of Naproche works similar to an include directive, which led to redundant checking of shared imports. Naproche-ZF tracks imports in a graph to avoid this. The import mechanism in Naproche-ZF uses the command “ $\backslash \text{import}\{<file>\}$ ” which may be hidden in the rendered document.

Proofs. The simplest way of proving a theorem in Naproche-ZF is to leave it entirely to the ATP by not writing an explicit proof. One can also provide a proof of the form “Follows by $\langle \text{justification} \rangle$ ”, where

$$\langle \text{justification} \rangle = \text{“set extensionality”} \mid \text{“assumption”} \mid \text{“definition”} \mid \langle \text{ref} \rangle.$$

The justification “by set extensionality” splits an atomic equation into two goals expressing mutual set inclusion, which is convenient when the ATP is reluctant about using extensionality. Next, “by assumption” and “by definition” each

restrict the available premises for the ATP to just the local assumptions or previous definition, respectively. Finally, a *<ref>* is an explicit reference to previous theorems, reusing commonly used L^AT_EX citation commands, such as `\cref` from the `Cleveref` package. Only these explicitly cited theorems are then used as premises for the ATP, together with the local assumptions and relevant definitions. Most other proof steps can also be justified in this manner, but we will disregard justifications below for the sake of brevity.

Next, one can state intermediate claims using one of many equivalent phrases such as “We have Φ ” or “Thus Φ ”. This creates an ATP task for the claim and then adds the claim as an additional assumption for the remainder of the proof. Claims may be justified by a **subproof**.

One can also perform goal-directed proof steps, similar to many other formalization languages. There are straightforward proof steps like “Assume Φ ”, “Suppose not”, and case distinctions. One can obtain witness with “Consider $x, y, z \sim X$ such that Φ ” or simplify universal goals with “Fix $x, y, z \sim X$ such that Φ ”, where both the bound by an arbitrary relation symbol \sim and the such-that refinement are optional.

A proof step of the form “It suffices to show Φ ” creates a proof task of showing that Φ implies the current goal, and then sets Φ as the new goal.

Naproche-ZF supports *calculational reasoning* in the `align*` environment: each equation may be followed by an `\explanation` with a citation. Currently calculational reasoning works with equations and biconditionals. Other systems such as Lean and Isabelle have similar features that also support inequalities [3]. We will extend calculational reasoning as needed alongside future formalizations.

Premise Selection. Irrelevant premises make it harder for ATPs to find proofs. *Premise selection* [24] is a process that attempts to identify the relevant premises in a problem. Naproche lacks premise selection, which is a major barrier to scaling beyond chapter-sized formalizations. Work is in progress to add premise selection to Naproche-ZF similar to the premise selection of Sledgehammer [23, 30]. The checker already includes a basic MePo-like [25] filter. There is also experimental premise selection using graph neural networks (GNNs), thanks to the help of Mirek Olšák and Josef Urban. The first-order problems exported by Naproche-ZF are structurally similar to the Mizar corpus on which GNN-based filters have performed well [14]. Training data for the GNN can be extracted from explicitly justified proof steps (those that use “by *<ref>*”). We also expect that premise selection trained on the Mizar corpus would perform well for Naproche-ZF. GNN-based premise selection is experimental and not yet integrated into the checker. We plan to scale up premise selection as we slowly grow the standard library.

Performance. An apples-to-apples performance comparison of Naproche and Naproche-ZF is difficult since there are no exactly parallel formalizations in the two systems. Naproche-ZF processes texts faster overall, in part due to having a parser with better asymptotic behaviour. The total checking time however is dominated by proof searches in external ATPs. ATP tasks are single-threaded in Naproche, whereas Naproche-ZF uses a thread pool to make use of modern

multi-core CPUs. Moreover, when an ATP task fails in Naproche, it retries the task after unfolding definitions. When developing large formalizations, this behaviour can lead to sudden explosions in checking time, as proofs that used to be fast suddenly become slow because they are retried multiple times. This behaviour dates back to SAD, where it was useful in the context of smaller formalizations and weaker ATPs. Naproche-ZF calls the ATP once per problem (but does use portfolio modes of ATPs). The standard library of Naproche-ZF is currently at a modest 4600 lines (excl. comments and blank lines). Using Vampire as the ATP, it takes less than 10s to check on an Intel i7-13700K and less than 22s on an Apple M1. In comparison, Naproche can take over 10 times as long when checking formalizations of similar length.

Naproche-ZF optionally caches the initial segment of successful ATP proofs between runs, resuming checking at the first failed proof, which saves time during proof writing. The cache of a proof is invalidated if the premises differ to avoid reproducibility problems: we do not use monotonicity of entailment since adding irrelevant premises can make ATP proofs fail.

4 Conclusion and Future Work

Even in its early state, Naproche-ZF is a new theorem prover that shows that natural theorem provers can scale beyond chapter-sized formalizations. It features an extensible grammar-based approach to natural language, familiar set-theoretical foundation in higher-order logic, and proof automation powered by strong first-order ATPs. Use of concurrency, more control over the proof search process, premise selection, faster parsing, a module system, and other refinements result in a performance improvement by an order of magnitude compared to its predecessor Naproche.

Naproche-ZF is still experimental research-quality software and requires more features, grammar refinement, bug fixes, user testing, and documentation to become user friendly software. Naproche-ZF's checker is a command line tool and lacks an integrated development environment (IDE), which would make the system more user friendly. Currently, user interaction with the ATP within Naproche-ZF is limited: failed or slow proofs sometimes require digging through large logs and experimenting with the ATP on the command line. An IDE for Naproche-ZF should also facilitate better interaction with the ATP, e.g. by giving Sledgehammer-like suggestions after finding proofs.

Naproche-ZF would also benefit from improvements to general purpose proof automation (e.g. better premise selection) and from including special-purpose proof automation, e.g. for arithmetic.

The included standard library is still fairly small and it would be nice to update student formalizations completed in older versions of Naproche to also work in Naproche-ZF.

Currently the \LaTeX files are typeset as-is. It would be worthwhile to generate richer HTML (with MathML) or PDF documents, by, e.g., linking lexical items to their definition or enabling progressive disclosure of more complicated proofs.

References

1. Abrahams, P.: The Proofchecker, MIT AI Memo (1961). <https://dspace.mit.edu/handle/1721.1/6068>
2. Bancerek, G., Naumowicz, A., Urban, J.: System description: XSL-based translator of Mizar to LaTeX. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) CICM 2018. LNCS (LNAI), vol. 11006, pp. 1–6. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96812-4_1
3. Bauer, G., Wenzel, M.: Computational reasoning revisited an Isabelle/Isar experience. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 75–90. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44755-5_7
4. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0 – the core of the TPTP language for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 491–506. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_41
5. Brown, C.E., Kaliszyk, C., Pał, K.: Higher-order Tarski–Grothendieck as a foundation for formal proof. In: Harrison, J., O’Leary, J., Tolmach, A. (eds.) ITP 2019 (2019)
6. Brown, C.E., Pał, K.: A tale of two set theories. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) CICM 2019. LNCS (LNAI), vol. 11617, pp. 44–60. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23250-4_4
7. Cheplyaka, R.: regex-applicative (2020). <https://hackage.haskell.org/package/regex-applicative>
8. De Lon, A., Koepke, P., Lorenzen, A.: A natural formalization of the mutilated checkerboard problem in Naproche. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). <https://doi.org/10.4230/LIPIcs.ITP.2021.16>, <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2021.16>
9. De Lon, A., Koepke, P., Lorenzen, A., Marti, A., Schütz, M., Wenzel, M.: The Isabelle/Naproche natural language proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 614–624. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_36
10. Détrez, G., Ranta, A.: Smart paradigms and the predictability and complexity of inflectional morphology. In: Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, pp. 645–653 (2012)
11. Earley, J.: An efficient context-free parsing algorithm. *Commun. ACM* **13**(2), 94–102 (1970). <https://doi.org/10.1145/362007.362035>
12. Farmer, W.M.: Formal mathematics for the masses. In: Workshop Papers of the 14th Conference on Intelligent Computer Mathematics (CICM 2021): Workshop on Natural Formal Mathematics (NatFoM 2021). CEUR Workshop Proceedings (2022). <https://ceur-ws.org/Vol-3377/natfom3.pdf>
13. Fredriksson, O.: Earley (2019). <https://hackage.haskell.org/package/Earley>
14. Goertzel, Z.A., Jakubův, J., Kaliszyk, C., Olšák, M., Piepenbrock, J., Urban, J.: The Isabelle ENIGMA. In: 13th International Conference on Interactive Theorem Proving (ITP 2022) (2022). <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.16>
15. Gordon, M.: Set theory, higher order logic or both? In: Goos, G., Hartmanis, J., van Leeuwen, J., von Wright, J., Grundy, J., Harrison, J. (eds.) TPHOLs 1996.

- LNCS, vol. 1125, pp. 191–201. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0105405>
16. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. *J. Formaliz. Reason.* **3**(2), 153–245 (2010)
 17. Harrison, J., Urban, J., Wiedijk, F.: History of Interactive Theorem Proving, vol. 9, pp. 135–214. North-Holland (2014). <https://doi.org/10.1016/B978-0-444-51624-4.50004-6>
 18. Hutton, G., Meijer, E.: Monadic parser combinators. *J. Funct. Program.* **8**(4), 437–444 (1998)
 19. Kmett, E.: Bound (2015). <https://www.schoolofhaskell.com/user/edwardk/bound>
 20. Kmett, E., Scott, R., Grenrus, O.: Bound: Making de Bruijn succ less (2023). <https://hackage.haskell.org/package/bound>
 21. Knuth, D.E.: Literate programming. Stanford University Center for the Study of Language and Information (1992)
 22. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
 23. Kühlwein, D., Blanchette, J.C., Kaliszyk, C., Urban, J.: MaSh: machine learning for sledgehammer. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 35–50. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_6
 24. Kühlwein, D., van Laarhoven, T., Tsvitsovadze, E., Urban, J., Heskes, T.: Overview and evaluation of premise selection techniques for large theory mathematics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 378–392. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_30
 25. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. *J. Appl. Log.* **7**(1), 41–57 (2009). <https://doi.org/10.1016/j.jal.2007.07.004>. <https://www.sciencedirect.com/science/article/pii/S1570868307000626>. Special Issue: Empirically Successful Computerized Reasoning
 26. Naproche contributors: Naproche. <https://naproche.github.io/>
 27. Paskevich, A.: Méthodes de formalisation des connaissances et des raisonnements mathématiques: aspects appliqués et théoriques. Ph.D. thesis, Université Paris 12 (2007)
 28. Paskevich, A.: The syntax and semantics of the ForTheL language (2007)
 29. Paulson, L.C.: ALEXANDRIA: large-scale formal proof for the working mathematician (2018). <https://www.cl.cam.ac.uk/~lp15/Grants/Alexandria/>
 30. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL 2010. The 8th International Workshop on the Implementation of Logics. EPIc Series in Computing, vol. 2, pp. 1–11. EasyChair (2012). <https://doi.org/10.29007/36dt>, <https://easychair.org/publications/paper/wV>
 31. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011)
 32. Ranta, A.: Some remarks on pragmatics in the language of mathematics: comments to the paper “at least one black sheep: Pragmatics and mathematical language” by luca san mauro, marco ruffino and giorgio venturi. *J. Pragmat.* **160**, 120–122 (2020). <https://doi.org/10.1016/j.pragma.2020.02.001>. <https://www.sciencedirect.com/science/article/pii/S0378216620300321>

33. Reger, G., et al.: Vampire 4.7-SMT system description. In: SMT-COMP 2022 (2022). <https://smt-comp.github.io/2022/system-descriptions/Vampire.pdf>
34. Rudnicki, P.: Obvious inferences. *J. Autom. Reasoning* **3**, 383–393 (1987). <https://doi.org/10.1007/BF00247436>
35. Ruffino, M., San Mauro, L., Venturi, G.: At least one black sheep: pragmatics and mathematical language. *J. Pragmat.* **160** (2020). <https://doi.org/10.1016/j.pragma.2020.01.011>
36. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
37. Sutcliffe, G., Suttner, C., Yemenis, T.: The TPTP problem library. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 252–266. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58156-1_18
38. Verchinine, K., Lyaletski, A., Paskevich, A.: System for Automated Deduction (SAD): a tool for proof verification. In: Automated Deduction–CADE-21, pp. 398–403 (2007)
39. Wiedijk, F.: The de Bruijn factor. Presented at a poster session at TPHOLs 2000 (2000). <https://www.cs.ru.nl/~freek/factor>
40. Wiedijk, F.: The QED manifesto revisited. In: From Insight to Proof, Festschrift in Honour of Andrzej Trybulec, pp. 121–133 (2007)

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

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





Reducibility Constraints in Superposition

Márton Hajdu¹  , Laura Kovács¹ , Michael Rawson¹ ,
and Andrei Voronkov^{2,3}

¹ TU Wien, Vienna, Austria

`marton.hajdu@tuwien.ac.at`

² University of Manchester, Manchester, UK

³ EasyChair, Manchester, UK

Abstract. Modern superposition inference systems aim at reducing the search space by introducing redundancy criteria on clauses and inferences. This paper focuses on reducing the number of superposition inferences with a single clause by blocking inferences into some terms, provided there were previously made inferences of a certain form performed with predecessors of this clause. Other calculi based on blocking inferences, for example basic superposition, rely on variable abstraction or equality constraints to express irreducibility of terms, resulting however in blocking inferences with all subterms of the respective terms. Here we introduce reducibility constraints in superposition to enable a more expressive blocking mechanism for inferences. We show that our calculus remains (refutationally) complete and present redundancy notions. Our implementation in the theorem prover Vampire demonstrates a considerable reduction in the size of the search space when using our new calculus.

Keywords: Saturation · Superposition · Redundancy · Reducibility constraints

1 Introduction

Automated reasoners in first-order logic with equality commonly rely on the *superposition calculus* [5, 25]. This calculus has been extended with various improvements in order to reduce the search space. For instance, avoiding superposition into variables and ordering literals and clauses are common practices in modern theorem provers [21, 29, 34].

To reduce generation of redundant clauses in equational reasoning, the “basicness” restriction [16] was introduced at the term level. This idea aided, for example, in finding the proof of the Robbins problem [24]. This restriction blocks superposition (rewriting) inferences into terms resulting from (quantifier) instantiations, considering such terms irreducible in further proof steps. This approach was further generalised to block superposition into terms above variable positions in basic superposition/paramodulation [7, 26], while preserving refutational completeness. However, blocking and applying different rewrite steps among equal

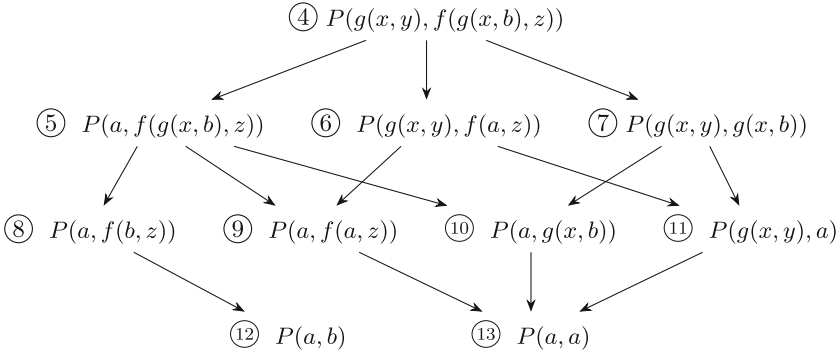


Fig. 1. Possible superposition sequences into 4.

terms impacts proof search. In this paper, we propose a number of different ways to block inferences, so that the resulting calculus remains complete. The effect of these restrictions resembles some strategies from term rewriting, such as innermost and outermost strategies.

Motivating Example. Consider the following satisfiable set \mathcal{C} of clauses:

$$\mathcal{C} = \left\{ \begin{array}{l} \textcircled{1} g(x, b) \simeq a, \quad \textcircled{2} f(x, b) \simeq x, \\ \textcircled{3} g(a, x) \simeq x, \quad \textcircled{4} P(g(x, y), f(g(x, b), z)) \end{array} \right\}$$

where x, y are variables, a, b constants, f, g function symbols, and P is a predicate symbol. In this paper \simeq denotes equality. Figure 1 shows some derivations of $P(a, a)$ by consecutively superposing into 4 with 1 and 2. It also shows a derivation of $P(a, b)$ by superposing into 4 with 1, then with 3 and 2. Note that Fig. 1 contains many redundant clauses. For example, 4 is redundant w.r.t. 6 and 1, as it is a logical consequence of (smaller) 6 and 1. Similarly, 7 is redundant w.r.t. 11 and 1.

Many derivations of Fig. 1 could however be avoided by using a rewrite order between the inferences. For example, a *leftmost-innermost rewrite order* on inferences derives 13 along the path 4–5–9–13. Whenever we would deviate from the leftmost-innermost rewrite order when rewriting a term t , we enforce the order by requiring that any term t' that is to the left of or inside t is irreducible in further derivations. In other words, we block further inferences with t' . With such a restriction, we cannot rewrite $g(x, y)$ in clause 6, as $g(x, y)$ was to the left of the previously rewritten term $f(g(x, b), z)$. Hence, when using a leftmost-innermost rewrite upon in Fig. 1, 9 is only generated by the derivation path 4–5–9. Similarly, 11 cannot be derived from 7 but can be derived from 6.

Our Contributions. We introduce a new superposition calculus that enables various ways to block (rewrite) inferences during proof search. Key to our calculus are *reducibility constraints* to restrict the order of superposition inferences (Sect. 3). Our approach supports and generalizes, among others, the leftmost-

innermost rewrite orders mentioned in the motivating example by means of *irreducibility constraints*, allowing us to reduce the number of generated clauses. Furthermore, in our motivating example the derivation ⑤–⑧–⑫ of Fig. 1 is not needed for the following reason. By superposing into ② with ③, we derive $a \simeq b$, which makes one of ⑫ and ⑬ redundant w.r.t. the other. As ① was used to rewrite $g(x, b)$ in Fig. 1 and derive ⑤, we block superposition into $g(x, b)$ with all clauses except ① in further derivations. We express this requirement via a *one-step reducibility constraint* (Definition 1), resulting in the BLINC – BLocked INference Calculus. As such, BLINC is parameterized by a rewrite ordering and (ir)reducibility constraints.

We prove¹ that our BLINC calculus is refutationally complete, for which we use a model construction technique (Sect. 4) with new features introduced to take care of constraints. We extend our calculus with redundancy elimination (Sect. 5). When evaluating the BLINC calculus implemented in the Vampire theorem prover, our experiments show that reducibility constraints significantly reduce the search space (Sect. 6).

2 Preliminaries

We work in standard *first-order logic with equality*, where equality is denoted by \simeq . We use variables x, y, z, v, w and terms s, t, u, l, r , all possibly with indices. A term is *ground* if it contains no variables. A literal is an unordered pair of terms with polarity, i.e. an equality $s \simeq t$ or a disequality $s \not\simeq t$. We write $s \bowtie t$ for either an equality or a disequality. A *clause* is a multiset of literals. We denote clauses by B, C, D and denote by \square the *empty clause* that is logically equivalent to \perp .

An *expression* E is a term, literal or clause. We will also consider as expressions constraints and constrained clauses introduced later. An expression is called *ground* if it contains no variables. We write $E[s]$ to state that the expression E contains a distinguished occurrence of the term s at some position. Further, $E[s \mapsto t]$ denotes that this occurrence of s is replaced with t ; when s is clear from the context, we simply write $E[t]$. We say that t is a *subterm* of $s[t]$, denoted by $t \trianglelefteq s[t]$; and a *strict subterm* if additionally $t \neq s[t]$, denoted by $t \triangleleft s[t]$. A *substitution* σ is a mapping from variables to terms, such that the set of variables $\{x \mid \sigma(x) \neq x\}$ is finite. We denote substitutions by $\theta, \sigma, \rho, \mu, \eta$. The application of a substitution θ on an expression E is denoted $E\theta$. A substitution θ is called *grounding for an expression* E if $E\theta$ is ground. We denote the set of grounding substitutions for an expression E by $\text{GSubs}(E)$, that is $\text{GSubs}(E) = \{\theta \mid E\theta \text{ is ground}\}$. We denote the empty substitution by ε .

A substitution θ is more general than a substitution σ if $\theta\eta = \sigma$ for some substitution η . A substitution θ is a *unifier* of two terms s and t if $s\theta = t\theta$, and is a *most general unifier*, denoted $\text{mgu}(s, t)$, if for every unifier η of s and t , there exists a substitution μ s.t. $\eta = \theta\mu$. We assume that the most-general unifiers of terms are idempotent [2].

¹ detailed proofs are in the full version of this paper [15].

A binary relation \rightarrow over the set of terms is a *rewrite relation* if (i) $l \rightarrow r \Rightarrow l\theta \rightarrow r\theta$ and (ii) $l \rightarrow r \Rightarrow s[l] \rightarrow s[r]$ for any term l, r, s and substitution θ . The *reflexive and transitive closure* of a relations \rightarrow is denoted by \rightarrow^* . We write \leftarrow to denote the inverse of \rightarrow . Two terms are *joinable*, denoted by $s \downarrow t$, if there exists a term u s.t. $s \rightarrow^* u \leftarrow^* t$. A *rewrite system* R is a set of rewrite rules. A term l is *irreducible* in R if there is no r s.t. $l \rightarrow r \in R$. Joinability w.r.t. R will be denoted by $s \downarrow_R t$. A *rewrite ordering* is a strict rewrite relation. A *reduction ordering* is a well-founded rewrite ordering. In this paper we consider reduction orderings which are total on ground terms, that is they satisfy $s \triangleright t \Rightarrow s \succ t$; such orderings are also called *simplification orderings*.

We use the standard definition of a *bag extension* of an ordering [12]. An ordering \succ on terms induces an ordering on literals, by identifying $s \simeq t$ with the multiset $\{s, t\}$ and $s \not\simeq t$ with the multiset $\{s, s, t, t\}$, and using the bag extension of \succ . We denote this induced ordering on literals also with \succ . Likewise, the ordering \succ on literals induces the ordering on clauses by using the bag extension of \succ . Again, we denote this induced ordering on clauses also with \succ . The induced relations \succ on literals and clauses are well-founded (resp. total) if so is the original relation \succ on terms. In examples used in this paper, we assume a KBO simplification ordering with constant weight [19].

Many first-order theorem provers work with clauses [21, 29, 34]. Let S be a set of clauses. Often, systems *saturate* S by computing all logical consequences of S with respect to a sound inference system \mathcal{I} . The process of saturating S is called *saturation*. An inference system \mathcal{I} is a set of inference rules of the form

$$\frac{C_1 \quad \dots \quad C_n}{C},$$

where C_1, \dots, C_n are the *premises* and C is the *conclusion* of the inference. The inference rule is *sound* if its conclusion is the logical consequence of its premises, that is $C_1, \dots, C_n \models C$. The inference is *reductive* w.r.t. an ordering \succ if $C \succ C_i$, for some $i = 1, \dots, n$. An inference system \mathcal{I} is *sound* if all its inferences are sound. An inference system \mathcal{I} is *refutationally complete* if for every unsatisfiable clause set S , there is a derivation of the empty clause in \mathcal{I} . An interpretation I is a model of an expression E if E is true in I . A clause C that is false in an interpretation I is a *counterexample* for I . If a clause set contains a counterexample, then it also contain a minimal counterexample w.r.t. a simplification ordering \succ [6].

3 Reducibility Constraints

This section presents a new blocking calculus, called BLINC (BLOcked INference Calculus). This calculus uses specific constraints to block inferences.

Definition 1 (Constraints). Let l be a non-variable term and r a term. We call the expression $l \rightsquigarrow r$ a *one-step reducibility constraint*, and the expression $\lceil l$ an *irreducibility constraint*. A *constraint* is one of the two. \square

$$\begin{array}{l}
 (\text{Sup}_{\ni}) \frac{l \simeq r \vee C \mid \Pi \quad \underline{s[u]} \bowtie t \vee D \mid \Gamma}{(s[r] \bowtie t \vee C \vee D)\sigma \mid \Delta} \quad \text{where} \quad \begin{array}{l}
 (1) \ u \text{ is not a variable,} \\
 (2) \ \sigma = \text{mgu}(l, u), \\
 (3) \ t\sigma \not\prec s[u]\sigma, r\sigma \not\prec l\sigma, \\
 (4) \ \Delta = \Gamma\sigma \cup \{l\sigma \rightsquigarrow r\sigma\} \\
 \quad \cup \mathcal{B}_{\ni}(s[u]\sigma, l\sigma), \\
 (5) \ \text{the conclusion is not blocked,}
 \end{array} \\
 \\
 (\text{EqRes}_{\ni}) \frac{s \not\prec t \vee C \mid \Gamma}{C\sigma \mid \Gamma\sigma} \quad \text{where} \quad \begin{array}{l}
 (1) \ \sigma = \text{mgu}(s, t), \\
 (2) \ \text{the conclusion is not blocked,}
 \end{array} \\
 \\
 (\text{EqFac}_{\ni}) \frac{s \simeq t \vee \underline{u} \simeq \underline{w} \vee C \mid \Gamma}{(s \simeq t \vee t \not\prec w \vee C)\sigma \mid \Gamma\sigma} \quad \text{where} \quad \begin{array}{l}
 (1) \ \sigma = \text{mgu}(s, u), \\
 (2) \ t\sigma \not\prec s\sigma, w\sigma \not\prec t\sigma, \\
 (3) \ \text{the conclusion is not blocked.}
 \end{array}
 \end{array}$$

Fig. 2. The BLINC calculus

Now let us define the semantics of these constraints.

Definition 2 (Satisfied Constraints, Violated Constraints). Let R be a rewrite system. We say that R *satisfies* $l \rightsquigarrow r$ if $l \rightarrow r \in R$ and *satisfies* $\downarrow l$ if l is irreducible in R . We say that R *violates* a constraint if it does not satisfy it. \square

An expression $C \mid \Gamma$ is a *constrained clause*, where C is a clause and Γ a finite set of constraints. A constrained clause $C \mid \Gamma$ is true iff C is true. We denote constrained clauses \mathcal{C}, \mathcal{D} , possibly with indices.

Definition 3 (Blocked Constrained Clause, Blocked Inference). Let $\mathcal{C} = C \mid \Gamma$ be a constrained clause. We call the constraint $l \rightsquigarrow r \in \Gamma$ *active in* \mathcal{C} if $s \succ l$ for some term s in C . Likewise, we call $\downarrow l \in \Gamma$ *active in* \mathcal{C} if $s \succ l$ for some term s in C . We call \mathcal{C} *blocked* if either it contains two active constraints $l \rightsquigarrow r$ and $l \rightsquigarrow r'$ such that r and r' are not unifiable, or it contains two active constraints $l \rightsquigarrow r$ and $\downarrow l$. An inference is *blocked* if its conclusion is blocked. \square

Our superposition calculus BLINC uses constrained clauses and bans inferences with blocked conclusions. For that, we attach constraints to clauses, as follows.

Definition 4 (S-ordering). An *S-ordering* is a partial strict well-order \ni on terms that is stable under substitutions. We use the function \mathcal{B}_{\ni} defined below to attach constraints to clauses.

$$\mathcal{B}_{\ni}(s, l) := \{\downarrow u \mid u \in l, u \text{ is non-variable and } u \trianglelefteq s\}$$

\square

BLINC is shown in Fig. 2. We assume a literal selection function satisfying the standard condition on \succ and underline selected literals. The next example illustrates blocked BLINC inferences.

Example 1. We use the order \succ on terms as the S-ordering. A Sup_{\ni} inference of BLINC into ④ with ② from our motivating example from page 2 results in

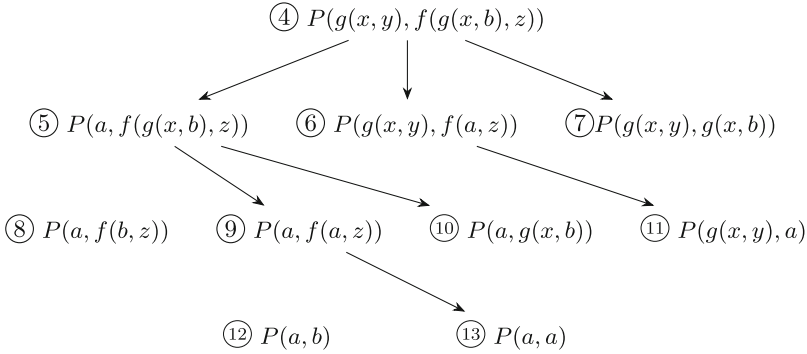


Fig. 3. Inferences from Fig. 1 with blocked inferences in BLINC removed. Figure 3 illustrates the effectiveness of reducibility constraints when compared to Fig. 1: we removed arcs corresponding to inferences blocked when the order \succ is used as the S-ordering. Of the 14 original inferences as in Fig. 1, only 7 are not blocked in Fig. 3.

$$\frac{f(x, b) \simeq x \quad P(g(x, y), f(g(x, b), z))}{P(g(x, y), g(x, b)) \mid \{\downarrow b, \downarrow g(x, y), \downarrow g(x, b), f(g(x, b), b) \rightsquigarrow g(x, b)\}}$$

Note that the conclusion is a constrained variant of clause (7) of Fig. 1. Now, the superposition of (1) into $g(x, y)$, and hence the following variant of clause (10) of Fig. 1, is blocked:

$$\frac{g(x, b) \simeq a \quad P(g(x, y), g(x, b)) \mid \{\downarrow g(x, y), \downarrow g(x, b), f(g(x, b), b) \rightsquigarrow g(x, b)\}}{P(a, g(x, b)) \mid \{\downarrow b, \downarrow g(x, b), f(g(x, b), b) \rightsquigarrow g(x, b), g(x, b) \rightsquigarrow a\}}$$

Note that the conclusion is blocked by the active constraints $g(x, b) \rightsquigarrow a$ and $\downarrow g(x, b)$. Figure 3 shows the modified version of Fig. 1, when using the inference rules of BLINC to generate fewer clauses than in Fig. 1. \square

Example 2. Consider now a sequence of superposition inferences into (4) by (1) and then by (3). That is, we consider the derivation (4)–(5)–(8) from Fig. 1 as:

$$\frac{g(x, b) \simeq a \quad P(g(x, y), f(g(x, b), z))}{g(a, x) \simeq x \quad P(a, f(g(x, b), z)) \mid \{\downarrow b, g(x, b) \rightsquigarrow a\}} \\ \frac{P(a, f(b, z)) \mid \{\downarrow a, \downarrow b, g(a, b) \rightsquigarrow a, g(a, b) \rightsquigarrow b\}}$$

The resulting conclusion is constrained and blocked, as we have two active constraints $g(a, b) \rightsquigarrow a$ and $g(a, b) \rightsquigarrow b$. As such and as shown in Fig. 3, clause (8) (and also clause (12)) will not be generated by BLINC, in contrast to Fig. 1. \square

4 Model Construction in BLINC

This section shows completeness of BLINC, with a proof which resembles that of Duarte and Korovin [13]. We start by adjusting terminology to our setting and discussing key differences compared with standard completeness proofs.

Definition 5 (Closure). Let $\mathcal{C} = C \mid \Gamma$ be a constrained clause and θ a substitution. The pair $\mathcal{C} \cdot \theta$ is called a *closure* and is logically equivalent to $C\theta$. A closure $(C \mid \Gamma) \cdot \theta$ is *ground* if $C\theta \mid \Gamma\theta$ is ground, in which case we say that θ is *grounding* for $C \mid \Gamma$ and call $(C \mid \Gamma) \cdot \theta$ a *ground instance* of $C \mid \Gamma$.

The set of all ground instances of \mathcal{C} is denoted \mathcal{C}^* . We will denote ground closures by \mathbb{C}, \mathbb{D} , maybe with indexes. If N is a set of constrained clauses, then N^* is defined as $\bigcup_{\mathcal{C} \in N} \mathcal{C}^*$. If $C \succ D$, we write $C \mid \Gamma \succ D \mid \Delta$. Similarly, if $C\theta \mid \Gamma\theta \succ D\sigma \mid \Delta\sigma$, then we write $(C \mid \Gamma) \cdot \theta \succ (D \mid \Delta) \cdot \sigma$. \square

A crucial part in the completeness proof of BLINC is reducing minimal counterexamples to smaller ones. However, due to blocked inference conditions (5) in Sup_{\supseteq} , (2) in EqRes_{\supseteq} , and (3) in EqFac_{\supseteq} , such a counterexample reduction may not be possible. We solve this problem in three steps:

1. Given a saturated set of clauses N , we construct a model for a subset of its closures $\mathcal{U}(N) \subseteq N^*$, namely, for so-called *unblocked closures* (Definition 6).
2. We show that if the empty clause \square is not in $\mathcal{U}(N)$, then the model satisfies each closure in $\mathcal{U}(N)$ (Theorem 1). That is, we show that counterexamples in $\mathcal{U}(N)$ can be reduced to smaller counterexamples that are also in $\mathcal{U}(N)$. This avoids the aforementioned problem with blocked inferences.
3. We then show that the model also satisfies all closures in N^* (Theorem 2).

Definition 6 (Partial/Total Models, Blocked/Productive Closures).

Let N be a set of constrained clauses. We will define, for every closure $\mathbb{C} \in N^*$, the rewrite system $R_{\mathbb{C}}$ and refer to all such rewrite systems as *partial models*. The definition will be made by induction on the relation \succ on ground closures. In parallel to defining $R_{\mathbb{C}}$, we also define the rewrite system

$$R_{\prec \mathbb{C}} = \bigcup_{\mathbb{D} \prec \mathbb{C}} R_{\mathbb{D}}.$$

The partial model $R_{\mathbb{C}}$ will either be the same as $R_{\prec \mathbb{C}}$, or obtained from $R_{\prec \mathbb{C}}$ by adding a single rewrite rule. In the latter case we call the closure \mathbb{C} *productive*.

The *reduced closure* of a ground closure $\mathcal{C} \cdot \theta$ is defined as the closure $\mathcal{C} \cdot \theta'$ such that for each variable x occurring in \mathcal{C} , we have that $\theta'(x)$ is the normal form of $\theta(x)$ in $R_{\prec \mathcal{C} \cdot \theta}$. We call a ground closure *reduced* if its reduced form coincides with this closure. Let $\mathcal{C} \cdot \theta$ be a ground closure and $\mathcal{C} \cdot \theta'$ be its reduced form. We say that $\mathcal{C} \cdot \theta$ is *blocked w.r.t. N* if $R_{\prec \mathcal{C} \cdot \theta'}$ violates some constraint in $\Gamma\theta'$ that is active in $\mathcal{C}\theta'$. A closure that is not blocked w.r.t. N is called *unblocked w.r.t. N* . Let $\mathcal{C} = (l \simeq r \vee C') \mid \Gamma$. The closure $\mathcal{C} \cdot \theta$ is called *productive* if

- (i) $\mathcal{C} \cdot \theta$ is false in $R_{\prec \mathcal{C} \cdot \theta}$,
- (ii) $l\theta \simeq r\theta$ is strictly maximal in $\mathcal{C}\theta$,
- (iii) $l\theta \succ r\theta$,
- (iv) $C'\theta$ is false in $R_{\prec \mathcal{C} \cdot \theta} \cup \{l\theta \rightarrow r\theta\}$,
- (v) $l\theta$ is irreducible in $R_{\prec \mathcal{C} \cdot \theta}$,
- (vi) $\mathcal{C} \cdot \theta$ is unblocked w.r.t. N .

Now we define

$$R_{\mathcal{C} \cdot \theta} = \begin{cases} R_{\prec \mathcal{C} \cdot \theta} \cup \{\!|\theta \rightarrow r\theta\!\}, & \text{if } \mathcal{C} \cdot \theta \text{ is productive;} \\ R_{\prec \mathcal{C} \cdot \theta}, & \text{otherwise.} \end{cases}$$

$$R_{\infty} = \bigcup_{\mathcal{C} \in N^*} R_{\mathcal{C}}$$

Finally, we call R_{∞} *the total model* and define $\mathcal{U}(N)$ as the set of all closures in N^* unblocked w.r.t. N . \square

This construction has two standard properties that we will use in our proofs:

1. $R_{\mathcal{C}} \models \mathbb{C}$ if and only if for all $\mathbb{D} \succ \mathbb{C}$ we have $R_{\mathbb{D}} \models \mathbb{C}$, if and only if $R_{\infty} \models \mathbb{C}$.
2. R_{∞} is non-overlapping, terminating and hence canonical.

The crucial difference between our model construction and the standard model construction is the condition on productive closures to be unblocked w.r.t. N . Let us now define our redundancy notions based on $\mathcal{U}(N)$ as follows.

Definition 7 (Redundant Clause/Inference). A constrained clause \mathcal{C} is *redundant w.r.t. N* if every ground instance of \mathcal{C} is either blocked w.r.t. N , or follows from smaller ground closures in $\mathcal{U}(N)$. An inference $\mathcal{C}_1, \dots, \mathcal{C}_n \vdash \mathcal{D}$ is *redundant w.r.t. N* if for each θ grounding for $\mathcal{C}_1, \dots, \mathcal{C}_n$ and \mathcal{D} either

- (i) one of $\mathcal{C}_1 \cdot \theta, \dots, \mathcal{C}_n \cdot \theta, \mathcal{D} \cdot \theta$ is blocked w.r.t. N , or
- (ii) $\mathcal{D} \cdot \theta$ follows from the set of ground closures $\{\mathcal{C} \mid \mathbb{C} \in \mathcal{U}(N) \text{ and } \mathbb{C}_i \cdot \theta \succ \mathbb{C} \text{ for some } i\}$. \square

Definition 8 (Saturation up to Redundancy). A set of constrained clauses N is *saturated up to redundancy* if, given non-redundant constrained clauses $\mathcal{C}_1, \dots, \mathcal{C}_n \in N$, any BLINC inference $\mathcal{C}_1, \dots, \mathcal{C}_n \vdash \mathcal{D}$ is redundant w.r.t. N . \square

From now on, let N be an arbitrary but fixed set of constrained clauses. We will formulate a sequence of lemmas used in the completeness proof, whose proofs are included in the full version of the paper [15]. The following lemma is used to show that unary inferences with an unblocked premise result in an unblocked conclusion.

Lemma 1. (Unblocking Inferences) *Suppose $\mathcal{C}, \mathcal{D} \in N$ and θ and σ are substitutions irreducible in $R_{\prec \mathcal{C} \cdot \theta}$ and in $R_{\prec \mathcal{D} \cdot \sigma}$, respectively. If $\mathcal{C} \cdot \theta \succ \mathcal{D} \cdot \sigma$, $\Gamma\theta \sqsupseteq \Delta\sigma$ and $\mathcal{C} \cdot \theta$ is unblocked w.r.t. N , then $\mathcal{D} \cdot \sigma$ is unblocked w.r.t. N .*

We next prove that the conclusion of a blocked inference is redundant, that is, the conditions that block inferences in BLINC are correct.

Lemma 2. (Redundancy with Blocked Clauses) *Let \mathcal{C} be a constrained clause. If \mathcal{C} is blocked, then all ground instances of \mathcal{C} are blocked w.r.t. N .*

The next lemma resembles the standard lemma on counterexample reduction.

Lemma 3 (Unblocked Sup_{\supseteq}). *Suppose that (a) $\mathcal{D} = s \bowtie t \vee D \mid \Gamma$ is a constrained clause in N , (b) $\mathcal{D} \cdot \theta$ a ground closure unblocked w.r.t. N , (c) θ is irreducible in $R_{\prec \mathcal{D} \cdot \theta}$, (d) $s\theta \succeq t\theta$, (e) $s\theta$ is reducible in $R_{\prec \mathcal{D} \cdot \theta}$.*

Then there exist a constrained clause $(\underline{l} \simeq \underline{r} \vee \mathcal{C} \mid \Pi) \in N$, a Sup_{\supseteq} -inference

$$(\text{Sup}_{\supset}) \frac{l \simeq r \vee C \mid \Pi \quad \underline{s[u] \bowtie t \vee D \mid \Gamma}}{(s[r] \bowtie t \vee C \vee D)\sigma \mid \Delta}$$

and a substitution τ such that (i) $\mathcal{D}\sigma\tau = \mathcal{D}\theta$, (ii) $l \simeq r \vee C \mid \Pi \cdot \sigma\tau$ is productive, and $(s[r] \bowtie t \vee C \vee D)\sigma \mid \Delta \cdot \sigma\tau$ is unblocked w.r.t. N .

We are now ready to show completeness of BLINC, starting with the following.

Theorem 1 (Model of $\mathcal{U}(N)$). *Let N be saturated up to redundancy and $\square \notin N$. Then for each $\mathbb{C} \in \mathcal{U}(N)$ we have $R_{\mathbb{C}} \models \mathbb{C}$.*

When $R_{\mathbb{C}} \models \mathbb{C}$, we will simply say that \mathbb{C} is true. Note that this implies that $R_{\mathbb{D}} \models \mathbb{C}$ for all $\mathbb{D} \succeq \mathbb{C}$, and also $R_{\infty} \models \mathbb{C}$. We say that \mathbb{C} is false if it not true.

Here, we only prove a few representative cases and refer to [15] for complete argumentation. Assume, by contradiction, that $\mathcal{U}(N)$ contains a ground closure \mathbb{C} such that $R_{\mathbb{C}} \not\models \mathbb{C}$. Since \succ is well-founded, then N^* contains a minimal unblocked closure $\mathcal{C} \cdot \theta$ such that $R_{\mathcal{C} \cdot \theta} \not\models \mathcal{C} \cdot \theta$.

Case 1. \mathcal{C} is redundant w.r.t. N .

Proof. The closure $\mathcal{C} \cdot \theta$ is unblocked, so it follows from smaller closures $\mathcal{C}_1, \dots, \mathcal{C}_n$ in $\mathcal{U}(N)$. Then there is some \mathcal{C}_i which is false too, and we are done. \square

Case 2. \mathcal{C} contains a variable x such that $x\theta$ is reducible in $R_{\prec \mathcal{C} \cdot \theta}$.

Proof. The reduced closure $\mathcal{C} \cdot \theta'$ of $\mathcal{C} \cdot \theta$ is unblocked w.r.t. N , so $\mathcal{C} \cdot \theta' \in \mathcal{U}(N)$. Since $x\theta \succ x\theta'$ and for all variables y different from x we have $y\theta \succeq y\theta'$, we have $\mathcal{C} \cdot \theta \succ \mathcal{C} \cdot \theta'$, then $\mathcal{C} \cdot \theta'$ is true. Since $y\theta = y\theta'$ is true in R_{∞} for all variables y , we also have that $\mathcal{C} \cdot \theta'$ is equivalent to $\mathcal{C} \cdot \theta$ in R_{∞} , hence $\mathcal{C} \cdot \theta$ is true and we obtain a contradiction. \square

Case 3. *There is a reductive inference $\mathcal{C}_1, \dots, \mathcal{C}_n \vdash \mathcal{D}$ with $\mathcal{C}_1, \dots, \mathcal{C}_n \in N$ which is redundant w.r.t. N such that (a) $\{\mathcal{C}_1 \cdot \theta, \dots, \mathcal{C}_n \cdot \theta\} \subseteq \mathcal{U}(N)$, (b) $\mathcal{D} \cdot \theta$ is unblocked w.r.t. N , (c) $\mathcal{C} \cdot \theta = \max\{\mathcal{C}_1 \cdot \theta, \dots, \mathcal{C}_n \cdot \theta\}$, and (d) $\mathcal{D} \cdot \theta \models \mathcal{C} \cdot \theta$.*

Proof. $\mathcal{D} \cdot \theta$ is implied by ground closures in $\mathcal{U}(N)$ smaller than $\mathcal{C} \cdot \theta$. These ground closures are then true in $R_{\mathcal{C} \cdot \theta}$, so $\mathcal{D} \cdot \theta$ is true, and hence $\mathcal{C} \cdot \theta$ is also true in $R_{\mathcal{C} \cdot \theta}$, contradiction. \square

Case 4. *None of the previous cases apply, and a negative literal $s \not\approx t$ is selected in \mathcal{C} , i.e. $\mathcal{C} = \underline{s \not\approx t \vee C \mid \Gamma}$.*

Proof. $\mathcal{C} \cdot \theta$ is false in $R_{\mathcal{C} \cdot \theta}$, so $s\theta \downarrow_{R_{\mathcal{C} \cdot \theta}} t\theta$. W.l.o.g., assume $s\theta \succeq t\theta$.

Subcase 4.1. $s\theta = t\theta$.

Proof. Then s and t are unifiable. Consider the EqRes_{\supset} inference

$$\frac{\underline{s \not\approx t \vee C \mid \Gamma}}{C\sigma \mid \Gamma\sigma}$$

where $\sigma = \text{mgu}(s, t)$. Take any ground instance $\mathcal{D} \cdot \rho = (C\sigma \mid \Gamma\sigma) \cdot \rho$ such that $\sigma\rho = \theta$; by the idempotence of σ , we have $\mathcal{D} \cdot \rho = \mathcal{D} \cdot \theta$. Clearly, $\mathcal{C} \cdot \theta \succ \mathcal{D} \cdot \theta$ and $\mathcal{D} \cdot \theta$ implies $\mathcal{C} \cdot \theta$. As $\mathcal{C} \cdot \theta \succ \mathcal{D} \cdot \theta$ and $\Gamma\sigma\rho = \Gamma\sigma\theta = \Gamma\theta$, Lemma 1 implies that $\mathcal{D} \cdot \theta$ is unblocked w.r.t. N . By Case 1, \mathcal{D} is not redundant, hence $\mathcal{D} \in N$. But then $\mathcal{D} \cdot \theta$ is a false closure in $\mathcal{U}(N)$, which is strictly smaller than $\mathcal{C} \cdot \theta$, so we obtain a contradiction. \square

Subcase 4.2. $s\theta \succ t\theta$.

Proof. By conditions on the literal selection, we assume that $s\theta \succ t\theta$ is maximal in \mathcal{C} . By Lemma 3, there is a Sup_{\supseteq} inference into $s\theta$ with a ground closure such that the result $\mathcal{C}' \cdot \theta$ is unblocked w.r.t. N . This closure is of the form $\mathcal{D} \cdot \theta = (l \simeq r \vee D \mid \Pi) \cdot \theta$ and we have the following Sup_{\supseteq} inference

$$\frac{l \simeq r \vee D \mid \Pi \quad \underline{s[l'] \not\prec t \vee C \mid \Gamma}}{(s[r] \not\prec t \vee C \vee D)\sigma \mid \Delta}$$

where $\sigma = \text{mgu}(l, l')$. Note that $\mathcal{C}' = s[r] \not\prec t \vee C \vee D$ and $\mathcal{C}' \cdot \rho = \mathcal{C}' \cdot \theta$. Then, $\mathcal{C} \cdot \theta \succ \mathcal{C}' \cdot \theta$ and $\mathcal{D} \cdot \theta$ and $\mathcal{C}' \cdot \theta$ imply $\mathcal{C} \cdot \theta$. Since $\mathcal{C}' \cdot \theta$ is unblocked w.r.t. N , using Lemma 2, we get that \mathcal{C}' is not blocked w.r.t. N , and condition (5) of Sup_{\supseteq} is satisfied. Similarly to Case 4.1, we have that the conclusion is a smaller false unblocked closure, so we obtain a contradiction. \square

Next we show that for a saturated set of clauses N , if R_{∞} is a model for $\mathcal{U}(N)$, then it is also a model of N^* , that is, R_{∞} satisfies also all blocked closures in N^* . This follows from the next theorem.

Theorem 2 (Model of N^*). *Let N be a saturated set of clauses. Every blocked closure $\mathcal{C} \cdot \theta \in N^*$ follows from $\mathcal{U}(N)$.*

Using Theorems 1–2, we obtain completeness of BLINC.

Corollary 1 (Completeness of BLINC). *Let N be saturated up to redundancy. If N does not contain \square , then N is satisfiable.*

We conclude with a remark on *constraint inheritance* in BLINC. Note that in the Sup_{\supseteq} inference rule of Fig. 2, constraints are inherited only from the right premise. It is possible to block more inferences without losing refutational completeness of BLINC, by allowing constraint inheritance from the left premise in the Sup_{\supseteq} rule as well. However, we cannot propagate constraints that are non-active in the left premise, as they may become active in the conclusion, making the inference blocked. This effect is illustrated in the following example.

Example 3. Consider a superposition into ① with ③

$$\frac{g(x, b) \simeq a \quad g(a, x) \simeq x}{a \simeq b \mid \{\downarrow a, \downarrow b, g(a, b) \rightsquigarrow a\}}$$

If $b \succ a$, then $\downarrow a$ is the only active constraint in the conclusion. Consider a superposition with ④ where constraints are inherited from both premises:

$$\frac{a \simeq b \mid \{\downarrow a, \downarrow b, g(a, b) \rightsquigarrow a\} \quad P(g(x, y), f(g(x, b), z))}{P(g(x, y), f(g(x, a), z)) \mid \{\downarrow a, \downarrow b, g(a, b) \simeq a, b \rightsquigarrow a\}}$$

In the conclusion, $\downarrow b$ and $b \rightsquigarrow a$ are both active, which blocks the inference. \square

5 Redundancy Detection in BLINC

In this section we discuss redundancy detection in BLINC. We give sufficient conditions for a clause to be redundant when inferences of a specific form are applied. As usual, we call a *simplifying inference*, or *simplification*, any inference such that one of the premises becomes redundant after the conclusion is added to the current set of clauses. Inference rules whose instances are simplifications are called *simplification rules*. When we display a simplification rule, we will denote clauses that become redundant by drawing a line through them.

Definition 7 gives rise to two kinds of simplification criteria: (i) based on blocking, and (ii) when one of the premises $\mathcal{C} \cdot \theta$ follows from smaller constrained clauses. The following definition captures the first redundancy criterion.

Definition 9 (Closure/Clause Blocked Relative to Closure/Clause).

A ground closure \mathbb{C} is *blocked relative to* a ground closure \mathbb{D} if for every set of constrained clauses N , if \mathbb{D} is blocked w.r.t. N^* , then \mathbb{C} is blocked w.r.t. N^* too. A constrained clause \mathcal{C} is *blocked relative to* a constrained clause \mathcal{D} , if every ground instance of \mathcal{C} is blocked relative to some ground instance of \mathcal{D} . \square

This notion will be used for defining simplification rules. We will next present sufficient conditions for checking that a constrained clause is blocked relative to another constrained clause. For example, each ground closure of a clause $C \mid \emptyset$ is unblocked w.r.t. any set N , hence everything is blocked relative to that ground closure. Further, each ground closure with a reducible substitution is blocked relative to its reduced closure.

Definition 10 (Well-Behaved Constrained Clause). Let $\mathcal{C} = C \mid \Gamma$ be a constrained clause. We say that \mathcal{C} is *well-behaved* if (i) all constraints in Γ are active in C , and for each $\gamma \in \Gamma$, (ii) if $\gamma = \downarrow l$, then $\downarrow u \in \Gamma$ for all $u \triangleleft l$, and (iii) if $\gamma = l \rightsquigarrow r$, then $\downarrow u \in \Gamma$ for all $u \triangleleft l$ and l contains all variables of r . \square

Example 4. The clause $P(a, f(b, z)) \mid \{\downarrow a, g(a, b) \rightsquigarrow a\}$ is not well-behaved but $P(a, f(b, z)) \mid \{\downarrow a, \downarrow b, g(a, b) \rightsquigarrow a\}$ is. The clause $a \simeq b \mid \{\downarrow a, \downarrow b, g(a, b) \rightsquigarrow a\}$ is not well-behaved since it contains constraints not active in the clause. \square

Lemma 4. (Relatively Blocked Well-Behavedness) Let $\mathcal{C} = C \mid \Gamma$ and $\mathcal{D} = D \mid \Delta$ be well-behaved constrained clauses, and σ a substitution. Then \mathcal{C} is blocked relative to \mathcal{D} if $\mathcal{C} \succ \mathcal{D}\sigma$ and $\Gamma \supseteq \Delta\sigma$.

In the sequel, we assume that each constrained clause is well-behaved. We next adjust two standard simplifications within superposition [14], namely demodulation in Theorem 3 and subsumption in Theorem 4. Our analogue of *demodulation* is the following special case of Sup_{\supseteq} in BLINC:

$$(\text{Dem}_{\supseteq}) \frac{l \simeq r \mid \Delta \quad \cancel{C[l\sigma] \mid \Gamma}}{C[r\sigma] \mid \Gamma} \quad \text{where} \quad \begin{array}{l} (1) l\sigma \succ r\sigma, \\ (2) C[l\sigma] \succ (l \simeq r)\sigma, \\ (3) \Delta\sigma \subseteq \Gamma. \end{array}$$

Theorem 3. (BLINC Demodulation) Dem_{\supseteq} is a simplification rule. That is, $C[l\sigma] \mid \Gamma$ is redundant w.r.t. any constrained clause set that contains $l \simeq r \mid \Delta$ and $C[r\sigma] \mid \Gamma$.

In addition to simplification rules, we will also consider *deletion rules*. These rules delete a (redundant) constrained clause from N provided that N contains another constrained clause or set of constrained clauses. The below deletion rule is our analogue of *subsumption*:

$$(\text{Subs}_{\supseteq}) \frac{D \mid \Delta \quad \cancel{C \mid \Gamma'}}{D\sigma \mid \Delta\sigma} \quad \text{where} \quad \begin{array}{l} (1) D\sigma \subsetneq C, \\ (2) \Delta\sigma \subseteq \Gamma, \end{array} \quad \text{for some substitution } \sigma.$$

Theorem 4. (BLINC Subsumption) Subs_{\supseteq} is a deletion rule. That is, $C \mid \Gamma$ is redundant w.r.t. any constrained clause set that contains $D \mid \Delta$.

We also introduce two deletion rules based on properties of the constraints of a clause. Namely, in Theorem 5 we introduce a deletion rule resembling “basic blocking” [25], whereas Theorem 6 exploits deletion based on rewrite orders. Consider therefore the following rule:

$$(\text{Block}_{\supseteq}) \frac{l \simeq r \mid \Delta \quad \cancel{C \mid \Gamma'}}{C \mid \Gamma} \quad \text{where} \quad \begin{array}{l} (1) C \succ (l \simeq r)\sigma \text{ and } l\sigma \succ r\sigma, \\ (2) \Delta\sigma \subseteq \Gamma, \\ (3) \text{either (i) } \downarrow l\sigma \in \Gamma \\ \text{or (ii) } l\sigma \rightsquigarrow r' \in \Gamma \text{ and } r' \succ r\sigma. \end{array}$$

Theorem 5. (BLINC Blocking) Block_{\supseteq} is a deletion rule. That is, $C \mid \Gamma$ is redundant w.r.t. any constrained clause that contains $l \simeq r \mid \Delta$.

Our last deletion inference relies on the fact that all rewrite rules in any partial model have to be oriented left-to-right according to \succ . That is,

$$(\text{Orient}_{\supseteq}) \frac{C \mid \Gamma \cup \{t \rightsquigarrow r\}}{C \mid \Gamma} \quad \text{where} \quad \begin{array}{l} (1) r \succ t, \\ (2) C \succ (t \simeq r). \end{array}$$

Theorem 6. (BLINC Orientation) $\text{Orient}_{\supseteq}$ is a deletion rule. That is, $C \mid \Gamma \cup \{t \rightsquigarrow r\}$ is redundant w.r.t. any constrained clause set.

We illustrate the above simplification and deletion rules with the following example.

Example 5. Consider the following well-behaved constrained clauses:

$$\begin{array}{ll} (1) P(g(a, x), b) \mid \{\downarrow b, f(x, b) \rightsquigarrow b\}, & (2) P(g(y, z), w) \mid \{f(z, w) \rightsquigarrow b\} \\ (3) g(a, z) \simeq b \mid \{\downarrow b\}, & (4) f(x, y) \simeq a \mid \emptyset \end{array}$$

By Theorem 4, clause (2) subsumes clause (1). By Theorem 3, clause (1) can be simplified with clause (3) into $P(b, b) \mid \{\downarrow b, f(x, b) \rightsquigarrow a\}$. Finally, assuming $b \succ a$, clauses (1) and (2) are redundant w.r.t. clause (4) by Theorem 5. \square

Remark 1. (Simplification Heuristics via Unblocking) We note that further simplifications (and heuristics) can be implemented by removing constraints from constrained clauses. This process of removing constraints is captured via the following rule:

$$(\text{Unblock}) \frac{C \not\vdash \Gamma}{C \mid \Delta} \text{ where } \Delta \subset \Gamma.$$

Clearly, **Unblock** is a simplification rule, as removing constraints from a constrained clause preserves completeness in **BLINC**. \square

We note that using the general notion of well-behaved clauses and Lemma 4, any further redundancy elimination technique can be adapted to **BLINC**. We conclude this section by showing that Theorems 3–6 can be adjusted and combined using the ground redundancy of Definition 7. This results in stronger redundancy detection, as the following example illustrates.

Example 6. Consider the following Sup_{\supseteq} inference:

$$\frac{g(f(v, w), a) \simeq g(w, a) \mid \emptyset \quad f(g(f(x, y), z), f(y, x)) \simeq z \mid \emptyset}{f(g(y, a), f(y, x)) \simeq a \mid \Delta} \sigma = \left\{ \begin{array}{l} v \mapsto x, \\ w \mapsto y, \\ z \mapsto a \end{array} \right\},$$

where $\Delta = \{\downarrow f(x, y), \downarrow f(y, x), \downarrow a, g(f(x, y), a) \rightsquigarrow g(y, a)\}$. Note that the conclusion is a well-behaved constrained clause. The conclusion cannot be simplified by clauses

$$(1) f(x, y) \simeq f(y, x) \quad \text{and} \quad (2) f(x, x) \simeq x,$$

using any of Theorems 3–6. However, using similar conditions as in the Block_{\supseteq} deletion rule, we can do the following. Let θ be a substitution that makes the conclusion ground. By a comparative case distinction on $x\theta$ and $y\theta$,

- (i) if $x\theta \succ y\theta$, then using clause (1), by $\downarrow f(x, y) \in \Delta$ and $f(x, y)\theta \succ f(y, x)\theta$;
- (ii) if $x\theta = y\theta$, then using clause (2) by $\downarrow f(x, y) \in \Delta$ (or $\downarrow f(y, x) \in \Delta$), $f(x, y)\theta = f(x, x)\theta \succ x\theta$ (or $f(y, x)\theta = f(x, x)\theta \succ x\theta$); and
- (iii) if $x\theta \prec y\theta$, then using clause (1) again, by $\downarrow f(y, x) \in \Delta$ and $f(y, x)\theta \succ f(x, y)\theta$;

we conclude that the ground closure $(f(g(y, a), f(y, x)) \simeq a \mid \Delta) \cdot \theta$ is redundant in all cases, hence the conclusion is redundant w.r.t. clauses (1) and (2). \square

Variant	UEQ		PEQ	
	Solved	Uniques	Solved	Uniques
baseline	778	15	1276	34
blinc1	316	0	411	0
blinc2	327	0	425	0
blinc3	610	0	809	0
blinc4	775	13	1270	28

Fig. 4. Experimental comparison using variants **BLINC** in Vampire, using 1455 UEQ problems and 2422 PEQ problems.

6 Evaluation

We implemented² BLINC in Vampire [21], together with the simplification rules of Sect. 5. We have also implemented a redundancy check called *orderedness* that eagerly checks if the result of a superposition can be deleted. We experimented with several variants of BLINC with redundancy elimination (all techniques discussed in Sect. 5), using different heuristics for removing constraints from clauses via Unblock: (i) `blinc1` does not use Unblock; (ii) `blinc2` uses Unblock to remove constraints inherited from premises, hence only conclusions of Sup_{\supset} will contain constraints; (iii) `blinc3` uses Unblock occasionally on the clause that would simplify the most clauses in the search space when unconstrained; (iv) `blinc4` uses Unblock on all clauses at activation. We compare these to standard superposition (`baseline`).

Solving unit equality (UEQ) problems is still very hard for superposition-based theorem provers, a claim substantiated by results in the CADE ATP System Competition (CASC) [30]. For this reason, our evaluation focused on the UEQ domain of the TPTP benchmark suite, version 8.1.2 [31]. Since our work does not consider (variants of) resolution, but proper superposition, we also restricted further evaluation to the pure equality (PEQ) benchmarks of TPTP. As a result, our experiments use all benchmarks of the unit equality (UEQ) and pure equality (PEQ) divisions from TPTP version 8.1.2 [31].

All our experiments are based on a DISCOUNT saturation loop [11] and a Knuth-Bendix ordering, with a timeout of 100s and without AVATAR [32]. Our results are summarized in Fig. 4. The results show that `blinc1` performs poorly compared to `baseline`, `blinc3` and `blinc4`, and that `blinc2` performs only slightly better than `blinc1`. The variant `blinc3` performs much better than `blinc1` and `blinc2` but it still does not solve any new problems. The variant `blinc4` performs comparably to the state-of-the-art baseline but solves different problems, 28 uniquely. Our preliminary results are therefore encouraging for complementing state-of-the-art superposition proving with BLINC reasoning, possibly in a portfolio solver.

We also analysed the impact of BLINC variants on skipping superposition inferences during proof search. Figure 5 shows the distribution of benchmarks by percentage of skipped superposition inferences among all superposition inferences during our runs for `blinc` variants. `blinc1` skips more than half of superposition inferences in a significant number of benchmarks, while the least restrictive `blinc4` still reduces the number of superposition inferences by a significant amount in most benchmarks.

² <https://github.com/vprover/vampire/commit/9c42b448996947e8>.

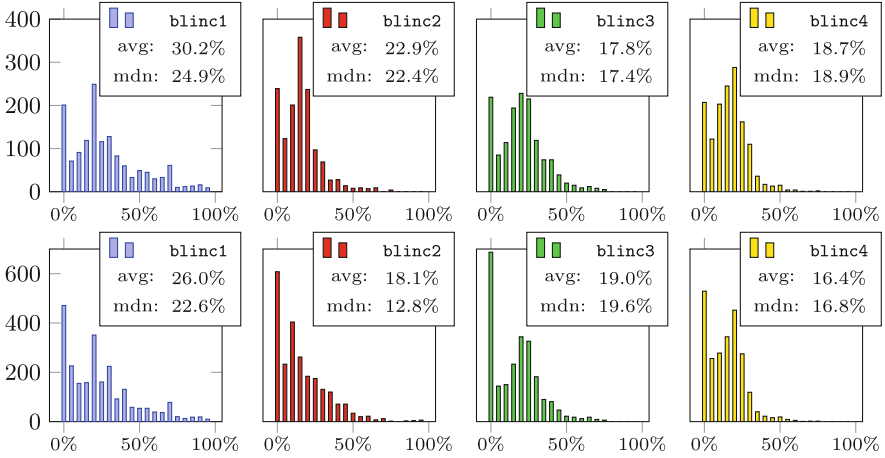


Fig. 5. Distribution of UEQ (top) and PEQ (bottom) benchmarks by ratio of skipped superpositions to all superpositions, showing also average (avg) and median (mdn). For example, using `blinc1`, on average 30.2%, resp. 26.0% of superpositions can be skipped in UEQ, resp. PEQ benchmarks.

7 Related Work

The basicness restriction [16,27] was extended to first-order logic, for example, in *basic superposition* [26] and *basic paramodulation* [7]. The former uses ground unification, the latter closures and variable abstraction to capture irreducibility constraints. In basic paramodulation, *redex orderings* are used similarly to S-orderings in our framework. BLINC expresses more fine-grained blocking, for example, distinguishing between different superpositions on the same term. Related notions in basic superposition have also been formalized [33].

Several critical pair criteria in completion-based theorem proving use irreducibility notions. *Blocking* [4] is similar to basicness, while *compositeness* [4,17] forbids any superpositions into terms with reducible subterms. *General superposition* [35,36] avoids superpositions when more general ones or ones symmetric in variables have been performed. Our BLINC framework handles all such restrictions. These criteria are instances of the *connectedness* criterion [3], which has been also explored in *ground joinability* [1], *ground reducibility* [22] and *ground connectedness* [13].

More general irreducibility constraints were considered in completion [23] and in superposition [18], the latter using a semantic tree method for completeness. Ordering constraints [9,10,20] and unification constraints [8,28] have also been considered, usually moving them to the calculus level. Extending and generalizing our BLINC framework with such constraints is a future challenge.

8 Conclusions

We introduce reducibility constraints to block inferences during superposition reasoning. Our resulting BLINC calculus is refutationally complete and is extended with redundancy elimination, allowing us to maintain efficient reasoning when compared to state-of-the-art superposition proving. Integrating our approach with further inference-blocking constraints, such as blocking more general or outermost superpositions, is an interesting line for future work. Adapting our framework to domain-specific inference rules, e.g. in linear arithmetic or higher-order superposition, is another line for future work.

Other interesting directions are (i) the use of a stronger semantics of constraints, as in Definition 10, and (ii) a “hybrid calculus”, improving on `blinc3`, where we still use constraints for blocking generating inferences but relax them whenever they prevent us from applying a simplification or a deletion rule.

Acknowledgements. We thank Konstantin Korovin for fruitful discussions. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

1. Avenhaus, J., Hillenbrand, T., Löchner, B.: On using ground joinable equations in equational theorem proving. *J. Symb. Comput.* **36**(1), 217–233 (2003). [https://doi.org/10.1016/S0747-7171\(03\)00024-5](https://doi.org/10.1016/S0747-7171(03)00024-5)
2. Baader, F., Nipkow, T.: Equational problems. In: *Term Rewriting and All That*, p. 58–92. Cambridge University Press (1998). <https://doi.org/10.1017/CBO9781139172752>
3. Bachmair, L.: *Canonical Equational Proofs*. Progress in theoretical computer science, Birkhäuser (1991). <https://doi.org/10.1007/978-1-4684-7118-2>
4. Bachmair, L., Dershowitz, N.: Critical pair criteria for completion. *J. Symb. Comput.* **6**(1), 1–18 (1988). [https://doi.org/10.1016/S0747-7171\(88\)80018-X](https://doi.org/10.1016/S0747-7171(88)80018-X)
5. Bachmair, L., Ganzinger, H.: Equational reasoning in saturation-based theorem proving. In: Bibel, W., Schmitt, P.H. (eds.) *Automated Deduction: A Basis for Applications*, vol. I, chap. 11, pp. 353–397. Springer (1998). <https://doi.org/10.1007/978-94-017-0437-3>
6. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: *Handbook of Automated Reasoning*, pp. 19–99. Elsevier and MIT Press (2001). <https://doi.org/10.1016/B978-044450813-3/50004-7>
7. Bachmair, L., Ganzinger, H., Lynch, C., Snyder, W.: Basic paramodulation and superposition. In: *CADE*, pp. 462–476 (1992). https://doi.org/10.1007/3-540-55602-8_185
8. Bhayat, A., Schoisswohl, J., Rawson, M.: Superposition with delayed unification. In: *CADE*, pp. 23–40 (2023). https://doi.org/10.1007/978-3-031-38499-8_2
9. Comon, H.: Solving symbolic ordering constraints. *Int. J. Found. Comput. Sci.* **01**(04), 387–411 (1990). <https://doi.org/10.1142/S0129054190000278>
10. Comon, H., Nieuwenhuis, R., Rubio, A.: Orderings, AC-theories and symbolic constraint solving (extended abstract). In: *Annual IEEE Symposium on Logic in Computer Science*, pp. 375–385 (1995). <https://doi.org/10.1109/LICS.1995.523272>

11. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT - a distributed and learning equational prover. *J. Autom. Reason.* **18**(2), 189–198 (1997). <https://doi.org/10.1023/A:1005879229581>
12. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. *Commun. ACM* **22**(8), 465–476 (1979). <https://doi.org/10.1145/359138.359142>
13. Duarte, A., Korovin, K.: Ground joinability and connectedness in the superposition calculus. In: *IJCAR*, pp. 169–187 (2022). https://doi.org/10.1007/978-3-031-10769-6_11
14. Gleiss, B., Kovács, L., Rath, J.: Subsumption demodulation in first-order theorem proving. In: *IJCAR*, pp. 297–315 (2020). https://doi.org/10.1007/978-3-030-51074-9_17
15. Hajdu, M., Kovács, L., Rawson, M., Voronkov, A.: Reducibility constraints in superposition. *EasyChair Preprint no. 12142* (EasyChair, 2024)
16. Hullot, J.M.: Canonical forms and unification. In: *CADE*, pp. 318–334 (1980). https://doi.org/10.1007/3-540-10009-1_25
17. Kapur, D., Musser, D.R., Narendran, P.: Only prime superpositions need be considered in the knuth-bendix completion procedure. *J. Symb. Comput.* **6**(1), 19–36 (1988). [https://doi.org/10.1016/S0747-7171\(88\)80019-1](https://doi.org/10.1016/S0747-7171(88)80019-1)
18. Kirchner, C., Kirchner, H., Rusinowitch, M.: *Deduction with Symbolic Constraints*. Research Report RR-1358, INRIA (1990)
19. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pp. 342–376. Springer (1983). https://doi.org/10.1007/978-3-642-81955-1_23
20. Korovin, K., Voronkov, A.: Knuth-bendix constraint solving Is NP-complete. In: *Automata, Languages and Programming*. pp. 979–992 (2001). https://doi.org/10.1007/3-540-48224-5_79
21. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: *CAV*, pp. 1–35 (2013). https://doi.org/10.1007/978-3-642-39799-8_1
22. Löchner, B.: A redundancy criterion based on ground reducibility by ordered rewriting. In: *IJCAR*, pp. 45–59 (2004). https://doi.org/10.1007/978-3-540-25984-8_2
23. Lynch, C., Snyder, W.: Redundancy criteria for constrained completion. In: *RTA*, pp. 2–16 (1993). https://doi.org/10.1007/978-3-662-21551-7_2
24. McCune, W.: Solution of the robbins problem. *J. Autom. Reason.* **19**, 263–276 (1997). <https://doi.org/10.1023/A:1005843212881>
25. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. in: *handbook of automated reasoning*, vol. I, chap. 7, pp. 371–443. Elsevier and MIT Press (2001). <https://doi.org/10.1016/B978-044450813-3/50009-6>
26. Nieuwenhuis, R., Rubio, A.: Basic Superposition is Complete. In: *ESOP*, pp. 371–389 (1992). https://doi.org/10.1007/3-540-55253-7_22
27. Nutt, W., Réty, P., Smolka, G.: Basic narrowing revisited. *J. Symb. Comput.* **7**(3–4), 295–317 (1989). [https://doi.org/10.1016/S0747-7171\(89\)80014-8](https://doi.org/10.1016/S0747-7171(89)80014-8)
28. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: *TACAS*, pp. 3–22 (2018). https://doi.org/10.1007/978-3-319-89960-2_1
29. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: *CADE*, pp. 495–507 (2019). https://doi.org/10.1007/978-3-030-29436-6_29
30. Sutcliffe, G.: The CADE ATP system competition - CASC. *AI Mag.* **37**(2), 99–101 (2016)
31. Sutcliffe, G.: The logic languages of the TPTP world. *Logic J. IGPL* (2022). <https://doi.org/10.1093/jigpal/jzac068>

32. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: CAV, pp. 696–710 (2014). https://doi.org/10.1007/978-3-319-08867-9_46
33. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. *J. Autom. Reason.* **66**(4), 499–539 (2022). <https://doi.org/10.1007/S10817-022-09621-7>
34. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischniewski, P.: SPASS Version 3.5. In: CADE, pp. 140–145 (2009). https://doi.org/10.1007/978-3-642-02959-2_10
35. Zhang, H., Kapur, D.: Consider only general superpositions in completion procedures. In: RTA, pp. 513–527 (1989). https://doi.org/10.1007/3-540-51081-8_129
36. Zhang, H., Kapur, D.: Unnecessary inferences in associative-commutative completion procedures. In: *Mathematical Systems Theory* (1990). <https://doi.org/10.1007/BF02090774>

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

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





First-Order Automatic Literal Model Generation

Martin Bromberger¹, Florent Krasnopol^{1,2}, Sibylle Möhle¹,
and Christoph Weidenbach¹(✉)

¹ Max Planck Institute for Informatics, Saarbrücken, Germany
{mbromber, fkrasnop1, smoehle, weidenbach}@mpi-inf.mpg.de

² École Normale Supérieure, Paris-Saclay, France

Abstract. Given a finite consistent set of ground literals, we present an algorithm that generates a complete first-order logic interpretation, i.e., an interpretation for all ground literals over the signature and not just those in the input set, that is also a model for the input set. The interpretation is represented by first-order linear literals. It can be effectively used to evaluate clauses. A particular application are SCL stuck states. The SCL (Simple Clause Learning) calculus always computes with respect to a finite number of ground literals. It then finds either a contradiction or a stuck state being a model with respect to the considered ground literals. Our algorithm builds a complete literal interpretation out of such a stuck state model that can then be used to evaluate the clause set. If all clauses are satisfied an overall model has been found. If it does not satisfy some clause, this information can be effectively explored to extend the scope of ground literals considered by SCL.

1 Introduction

Explicit and effective model representations as well as model building out of a set of first-order clauses have a long tradition [3, 10, 12, 16, 20, 24, 32, 41, 48, 50, 51]. In addition, they naturally arise out of decision procedures for decidable first-order clause set fragments [1–3, 11, 14, 15, 18, 19, 22, 25–31, 33, 36, 38, 39, 43, 44, 46, 52–55]. The problem we are studying here is to the best of our knowledge new: given a finite set of consistent ground literals, find a finite representation of an overall, typically infinite Herbrand style interpretation, satisfying those ground literals. Of course, there are trivial solutions to this problem, e.g., by assigning any missing ground literal to true or false. Projecting the results of [23] to first-order logic results in such a trivial solution. However such a solution will not fit our motivating application: the family of SCL calculi [6, 7, 9, 21, 37] where we here concentrate on the case of first-order logic without equality. Similar to CDCL [40], SCL computes resolution inferences with respect to a partial ground model, i.e., a consistent sequence of first-order ground literals. The number of ground literals considered by SCL is finite at any point in time, thanks to an upper bound ground literal β with respect to a well-founded (quasi)-ordering. For the purpose of this paper we simply consider the number of symbols in a

literal with respect to \leq . In this context SCL either produces the empty clause with respect to β or a partial model satisfying all first-order clause instances smaller than β . In case of such a partial model we want to extend it to an overall interpretation for the clause set and then check whether this interpretation is a model for the first-order clause set considered, or, if not, find a suitable extension to β that then covers false clauses with respect to the generated interpretation. So all considered ground literals are instances of existing literals from some clause set. Therefore, we look for a solution that respects the term structure of the ground literals. Our approach starts with a universal relation and then refines it according to the term structure of the considered ground literals until it fits all ground literals,

For illustration, consider the following very simple example. For the three first-order clauses

$$\begin{array}{l} \neg P(x) \vee P(g(g(x))) \\ \neg P(g(g(g(a)))) \end{array} \quad P(a)$$

an SCL run with $\beta = P(g(g(g(a))))$, i.e., exclusively atoms smaller or equal $P(g(g(g(a))))$ are dealt with, where for the ordering we simply count symbols, a partial model generated by SCL could be

$$\{P(a), P(g(g(a))), P(g(a))^1, P(g(g(g(a))))\}$$

where the third literal is decided and the others are propagated from the above clauses. It is a model for all ground instances with literals smaller or equal $P(g(g(g(a))))$, hence, excluding $\neg P(g(g(g(a))))$. Our model building calculus would start with state

$$(\{P(a), P(g(g(a))), P(g(a)), P(g(g(g(a))))\}; \emptyset; \{P(x)\}; \emptyset)$$

meaning that the initial model assumption for P is the universal relation, i.e., P holds on all ground terms. Processed ground literals are moved from the first to the second component of the state and final literal interpretation literals from the third to the fourth component by the algorithm. Thus finally, all processed ground literals are moved to the second component and the final literal model is contained in the fourth component while the other two are empty. One application of rule Solve, see page 7, immediately establishes the model $P(x)$, because it satisfies all ground literals. Of course, this interpretation does not satisfy the three clauses without the restriction to instances bounded by β . Still, we can use our interpretation to find the smallest clause instance falsified by it, in our example $\neg P(g(g(g(a))))$, and use the maximal literal in that clause as our new $\beta = P(g(g(g(a))))$. Running SCL with the new β will immediately yield the contradiction. Now consider a small modification of the three clauses where we replace the final unit clause by a disjunction.

$$\begin{array}{l} \neg P(x) \vee P(g(g(x))) \\ \neg P(g(g(g(a)))) \vee \neg P(g(g(a))) \end{array} \quad P(a)$$

Running SCL on this clause set with $\beta = P(g(g(g(a))))$ may yield the same partial model as before and hence the same overall interpretation $P(x)$. Again

the final clause is falsified by the interpretation yielding a new minimal $\beta = P(g(g(g(g(a)))))$. Running SCL again with this β yields a final partial model

$$[P(a), P(g(g(a))), P(g(g(g(g(a))))), \neg P(g(g(g(a))))], \neg P(g(g(a)))^1]$$

and now starting with this ground model

$$(\{P(a), P(g(g(a))), P(g(g(g(g(a))))), \neg P(g(g(g(a))))\}; \emptyset; \{P(x)\}; \emptyset)$$

the initial candidate interpretation $P(x)$ needs to be refined, because it has positive and negative instances among the set of ground literals. Refining means, we exhaustively instantiate $P(x)$ until no model candidate atom has both positive and negative instances by rule Refine, see page 6. This eventually yields

$$(\emptyset; \{P(a), P(g(g(a))), P(g(g(g(g(a))))), \neg P(g(g(g(a))))\}; \emptyset; \{P(a), \neg P(g(a)), P(g(g(a))), \neg P(g(g(g(a))))\})$$

which in fact covers all ground literals and constitutes a model for the three clauses.

The paper is now organized as follows: after fixing some notions and notation in Sect. 2, and a short introduction to SCL, Sect. 3, our contributions are contained in Sect. 4. Important results are: (i) out of a set of ground literals we can generate in polynomial time an overall interpretation, Lemma 4, Lemma 8 and Lemma 5; (ii) our literal model representation satisfies the well-known requirements for explicit model representations [10], in particular supports effective clause evaluation, see page 13; (iii) the literal model representation can effectively be used to find a new minimal β in case it does not satisfy the clause set, Lemma 13. The paper ends with a discussion of the obtained results and further research directions, Sect. 5.

2 Preliminaries

We assume a first-order language without equality over a signature $\Sigma = (\Omega, \Pi)$ of operator symbols and predicates, respectively. All signature symbols come with a fixed arity. Terms, atoms, literals, clauses and clause sets are defined as usual, where in particular clauses are identified both as disjunctions and multisets of literals. Then N denotes a clause set; C, D denote clauses; L, K, H denote literals; A, B denote atoms; P, Q, R denote predicates; t, s terms; f, g, h function symbols; a, b, c constants; and x, y, z variables. We write $f/1$ or $R/2$ for a function symbol of arity 1 or predicate symbol of arity 2, respectively. The complement of a literal is denoted by the function comp. The atom of a literal by the function atom, i.e., $\text{atom}(\neg A) = A$ and $\text{atom}(A) = A$. Semantic entailment \models is defined as usual where variables in clauses are assumed to be universally quantified. Substitutions σ, τ are total mappings from variables to terms, where $\text{dom}(\sigma) := \{x \mid x\sigma \neq x\}$ is finite and $\text{codom}(\sigma) := \{t \mid x\sigma = t, x \in \text{dom}(\sigma)\}$. Their application is extended to literals, clauses, and sets of such objects in the

usual way. A term, atom, clause, or a set of these objects is *ground* if it does not contain any variable. A substitution σ is *ground* if $\text{codom}(\sigma)$ is ground. A substitution σ is *grounding* for a term t , literal L , clause C if $t\sigma$, $L\sigma$, $C\sigma$ is ground, respectively. A literal L is an *atom instance* of a literal K if $\text{atom}(K)\sigma = \text{atom}(L)$ for some σ . A term, literal is called *linear* if any variable occurs at most once in the term, literal. The function mgu denotes the *most general unifier* of two terms, atoms, literals. We assume that any mgu of two terms or literals does not introduce any fresh variables and is idempotent.

A *position* is a word over the naturals with empty word ϵ . The set of positions of a term, atom is inductively defined by: $\text{pos}(x) = \{\epsilon\}$ if x is a variable, $\text{pos}(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \bigcup_{i=1}^n \{ip \mid p \in \text{pos}(t_i)\}$ for terms, and $\text{pos}(P(t_1, \dots, t_n)) = \{\epsilon\} \cup \bigcup_{i=1}^n \{ip \mid p \in \text{pos}(t_i)\}$ for atoms. For a position $p \in \text{pos}(t)$ we define $t|_p = t$ if $p = \epsilon$ and $f(t_1, \dots, t_n)|_p = t_{i|p'}$ if $p = ip'$. Moreover, we define by $t[s]_p$ the term, atom one receives by replacing the subterm $t|_p$ at position p of t with the term s . The size of a term t , atom A is defined by $\text{size}(t) = |\text{pos}(t)|$ or $\text{size}(A) = |\text{pos}(A)|$. The size of a substitution σ is defined by $\text{size}(\sigma) = \sum_{x \in \text{dom}(\sigma)} \text{size}(x\sigma)$. The size of a set of terms, atoms, substitution is the sum of the size of its members. A position $p \in \text{pos}(t)$ is *maximal* in t if for any other position $q \in \text{pos}(t)$ we have $|q| \leq |p|$. The *depth* of a position p is 0 if $p = \epsilon$ and $|p|$ otherwise. The *depth* of a term t , atom A is the maximal depth of any position in t , A , i.e., $\text{depth}(t) = \max\{|p| \mid p \in \text{pos}(t)\}$ and $\text{depth}(A) = \max\{|p| \mid p \in \text{pos}(A)\}$, respectively. The depth of a term s in a term t is the depth of a maximal position p such that $t|_p = s$.

Two literals are *inconsistent* if they have different sign and their atoms are unifiable. A set of literals is *consistent* if it does not contain a pair of inconsistent literals. A *literal interpretation* M is a finite set of consistent literals. A literal interpretation \mathcal{I} is *complete* with respect to a signature Σ if for any Σ ground atom A there is a literal $K \in \mathcal{I}$ such that $\text{atom}(K)\sigma = A$ for some σ . A literal interpretation \mathcal{I} satisfies a ground literal K , $\mathcal{I} \models K$ if there is an $L \in \mathcal{I}$ such that $L\sigma = K$ for some σ . It satisfies a non-ground literal K if it satisfies all groundings of K .

We overload notation for sets where “,” is overloaded for disjoint union, and disjoint addition, e.g., “ Γ_1, Γ_2 ” stands for $\Gamma_1 \cup \Gamma_2$ and Γ_1, L stands for the set $\Gamma_1 \cup \{L\}$.

3 SCL: Clause Learning from Simple Models

The family of SCL calculi (short “Simple Clause Learning”) [6, 9, 21, 37] lifts CDCL (Conflict-Driven Clause Learning) from propositional logic [34, 42, 49] to variants of first-order logic. The idea is to have superposition-style resolutions on non-ground, first-order clauses but instead of the usual static order that guides them, SCL uses as its guide ground partial models Γ , i.e., sequences of ground literals, also called *trails*. A trail for a clause set N is constructed/extended by guessing literals via so called Decisions and by propagating literals based on the current trail and the current clauses in N [9]. This construction continues until

we determine that Γ falsifies a ground instance $C\sigma$ of a clause $C \in N$. The conflict between Γ and C is then resolved by applying Resolution to C and the clauses used for propagation during the construction of Γ . At the end of these resolutions, SCL learns a new clause D and a prefix Γ' of Γ from which D can be propagated to start the construction of the next trail, which is guaranteed to never encounter the same conflict due to D . Furthermore D is not redundant, in particular, not subsumed by any clause.

The maximal length of the trail is always finitely bounded by all literals being smaller than a fixed ground literal β . In case all ground literals have been explored and not clause is falsified this constitutes a so called *stuck state*. In a stuck state the trail is model for all ground clause instantiations smaller β , but not in general.

In its first, original version [21], the focus of the SCL calculus is on deciding the Bernays-Schoenfinkel class without equality. Moreover, the original version is already a sound and refutationally complete semi-decision procedure for general first-order logic without equality that guarantees non-redundant clause learning. Subsequently, SCL has been extended to handle theories [6] and first-order logic with equality [37].

In the meantime, there exists a refined version [9] unifying and extending the previous versions [6,37] for first-order logic called SCL(FOL). In particular, this version introduces a refined Backtrack rule and a refined reasonable strategy criterion. In parallel we proved correctness and soundness of SCL(FOL) in Isabelle [5]. The Isabelle SCL(FOL) version relaxes some of the original requirements. SCL computations are performed with respect to a quasi-ordering \preceq on ground atoms where the strict part is well-founded. We adopt this setting also in this paper by instantiating \preceq with symbol counting and \leq . SCL(FOL) is only allowed to add literals L to the trail Γ with $\text{atom}(L) \preceq \beta$ for some atom β . Note that the bounding atom β may grow, but only if we reach a stuck state, where $\Gamma \models \text{gnd}^{\preceq\beta}(N)$ and where the function $\text{gnd}^{\preceq\beta}$ computes the set of all ground instances of a clause set where the grounding is restricted to produce literals L with $\text{atom}(L) \preceq \beta$. This guarantees that SCL(FOL) (with a reasonable strategy) will always find a refutation if the input clause set is unsatisfiable. Moreover, for a fixed β , SCL(FOL) turns into a decision procedure for $\text{gnd}^{\preceq\beta}(N)$. And even if we allow β to grow, SCL(FOL) regularly visits partial models Γ that at least satisfy $\text{gnd}^{\preceq\beta}(N)$, that may even be extendable to full models for N , or at least guide the selection for our next bounding literal β' [8].

4 Generating Models

A motivation for our model generating algorithm is the extension of SCL ground trail Γ out of a stuck state to a complete literal interpretations. Such an interpretation either satisfies the considered clause set, or it falsifies some clause. The latter information can then be used to extend the SCL search for a model or a contradiction. Our extension from Γ is not trivial, e.g., by assigning all atoms beyond Γ to true. Instead, it respects the literal structure in Γ and naturally extends it to a complete literal interpretation.

The starting point is simply a set of ground literals and the finite signature used to build the set. The algorithm is presented by three abstract rewrite rules operating on a state in a non-deterministic way. The state is a tuple $(\Gamma; \Delta; \mathcal{I}; M)$ where Γ , Δ are consistent sets of ground literals, M is a set of linear literals that defines a partial interpretation such that $M \models L$ for each $L \in \Delta$ and M does not have any conflict with Γ ; \mathcal{I} is a set of linear atoms such that $\mathcal{I} \cup M$ represents a complete literal interpretation; initially M is empty and \mathcal{I} the set $\{P(x_1, \dots, x_n)\}$ for some predicate P and linear atom $P(x_1, \dots, x_n)$, denoting the universal relation for P . Processed literals/atoms are moved by the rewrite rules from Γ to Δ and \mathcal{I} to M , respectively. The rewrite calculus then builds an overall interpretation of P according to Γ where we assume that Γ only contains P literals. So given a set of ground literals for each occurring predicate a separate run starting with the respective literals is needed.

The start state is $(\Gamma; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$ for a finite consistent set of ground literals Γ over P and linear atom $P(x_1, \dots, x_n)$ and a final state is $(\emptyset; \Delta; \emptyset; M)$ where we will show $M \models \Gamma$. We assume a finite signature Σ .

The first rule Refine covers the situation where some atom A in \mathcal{I} has both positive and negative instances in Γ . Since Γ is consistent, the atom A can be split into instances A_i of itself and each of the resulting instances is guaranteed to eventually have only positive or negative instances in Γ . Note that this may require repeated applications of the rule Refine.

Refine $(\Gamma; \Delta; \mathcal{I}, P(t_1, \dots, t_n); M)$
 $\Rightarrow_{\text{mod}} (\Gamma; \Delta; \mathcal{I}, \cup_{f_i/k_i \in \Omega} \{P(t_1, \dots, t_n)\{x \mapsto f_i(y_{i_1}, \dots, y_{k_i})\}\}; M)$

provided $P(t_1, \dots, t_n)|_p = x$, f_i are all function symbols (including constants) from Ω and k_i denotes their respective arity, all variables in $f_i(y_{i_1}, \dots, y_{k_i})$ are fresh, different variables, and p is a minimal variable position in $P(t_1, \dots, t_n)$ for which there exist literals $P(l_1, \dots, l_n)$ and $\neg P(r_1, \dots, r_n)$ in Γ such that both are atom instances of $P(t_1, \dots, t_n)$, and the atoms $P(l_1, \dots, l_n)|_p$ and $P(r_1, \dots, r_n)|_p$ are not unifiable

Due to refinement and the construction of the final complete literal interpretation it may happen that certain atoms in \mathcal{I} do not have any instances in Γ . They are then moved to the final representation of the interpretation by rule Clean.

Clean $(\Gamma; \Delta; \mathcal{I}, P(t_1, \dots, t_n); M)$
 $\Rightarrow_{\text{mod}} (\Gamma; \Delta; \mathcal{I}; M, P(t_1, \dots, t_n))$

provided $P(t_1, \dots, t_n)$ has no atom instance in Γ

Actually, the atom $P(t_1, \dots, t_n)$ could be added positively or negatively to the final literal interpretation. In favor of Theorem 12 we stick here to adding all literals without instances positively. If all instances of some atom in \mathcal{I} from Γ have an identical sign, they are solved and both the atom and the instances can be removed from \mathcal{I} and Γ by rule Solve, respectively.

Solve $(\Gamma; \Gamma'; \Delta; \mathcal{I}, P(t_1, \dots, t_n); M)$
 $\Rightarrow_{\text{mod}} (\Gamma; \Delta, \Gamma'; \mathcal{I}; M, \#P(t_1, \dots, t_n))$

provided Γ' , $\Gamma' \neq \emptyset$, consists only of positive literal instances of $P(t_1, \dots, t_n)$ or only of negative literal instances; Γ contains no literal instances of $P(t_1, \dots, t_n)$; $\# = \neg$ if the literals in Γ' are negative and $\#$ is empty otherwise

Example 1. Let $\Sigma = (\{a/0, f/1, g/1\}, \{P/3\})$ be a signature. Now consider $\Gamma = \{K, L\}$ where $K = P(a, f(a), a)$, $L = \neg P(a, g(a), a)$ over the signature Σ . An execution trace of \Rightarrow_{mod} is as follows:

- 1 : $(\{K, L\}; \emptyset; \{P(x_1, x_2, x_3)\}; \emptyset)$
- 2 : $\Rightarrow_{\text{mod}}^{\text{Refine}} (\{K, L\}; \emptyset; \{P(x_1, a, x_3), P(x_1, f(y_1), x_3), P(x_1, g(y_2), x_3)\}; \emptyset)$
- 3 : $\Rightarrow_{\text{mod}}^{\text{Clean}} (\{K, L\}; \emptyset; \{P(x_1, f(y_1), x_3), P(x_1, g(y_2), x_3)\}; \{P(x_1, a, x_3)\})$
- 4 : $\Rightarrow_{\text{mod}}^{\text{Solve}} (\{L\}; \{K\}; \{P(x_1, g(y_2), x_3)\}; \{P(x_1, a, x_3), P(x_1, f(y_1), x_3)\})$
- 5 : $\Rightarrow_{\text{mod}}^{\text{Solve}} (\emptyset; \{K, L\}; \emptyset; \{P(x_1, a, x_3), P(x_1, f(y_1), x_3), \neg P(x_1, g(y_2), x_3)\})$

Step 1: The initial state $(\Gamma; \Delta; \mathcal{I}; M)$ consists of the set $\Gamma = \{K, L\}$, the empty set Δ , the set \mathcal{I} containing only $P(x_1, x_2, x_3)$ which generalizes all literals over Σ with predicate P , and the empty set of literals M .

Step 2: Both K and L are atom instances of $P(x_1, x_2, x_3)$, but with opposite signs. Moreover, the terms $K|_2 = f(a)$ and $L|_2 = g(a)$ are not unifiable, and the position $p = 2$ is the minimal variable position in $P(x_1, x_2, x_3)$ for which this is the case, and the preconditions of rule Refine are met. A refinement of $P(x_1, x_2, x_3)$ in position 2 takes place and $P(x_1, x_2, x_3)$ is replaced by literals differing from it only in position 2 by replacing x_2 by every constant and function symbol occurring in Σ . The resulting atoms are $P(x_1, a, x_3)$, $P(x_1, f(y_1), x_3)$, and $P(x_1, g(y_2), x_3)$ and they cover again all P ground instances.

Step 3: The literal $P(x_1, a, x_3) \in \mathcal{I}$ has no atom instance on Γ and is moved to M by means of rule Clean.

Step 4: The positive literal K is the only instance of $P(x_1, f(y_1), x_3)$ on Γ , and the preconditions of rule Solve are met. The literal K is moved from Γ to Δ , and $P(x_1, f(y_1), x_3)$ is moved from \mathcal{I} to M .

Step 5: The negative literal L is the only atom instance of $P(x_1, g(y_2), x_3)$ with negative sign, and the preconditions of rule Solve are met. The literal L is moved from Γ to Δ , whereas $P(x_1, g(y_2), x_3)$ is removed from \mathcal{I} and added to M with negative sign. Now both Γ and \mathcal{I} are empty, and the execution stops with the linear complete literal interpretation $M = \{P(x_1, a, x_3), P(x_1, f(y_1), x_3), \neg P(x_1, g(y_2), x_3)\}$.

Next we prove that \Rightarrow_{mod} always computes an overall interpretation and model of the initial Γ . The basis for these results is the notion of a sound state below. We then show by induction on the length of a \Rightarrow_{mod} derivation that the initial state is sound and any follower state is sound assuming its start state is sound.

Definition 2 (Sound State). *A state $(\Gamma; \Delta; \mathcal{I}; M)$ is sound, if the following invariants hold:*

1. All literals in $\mathcal{I} \cup M$ are linear.
2. The atoms of any two different literals from $\mathcal{I} \cup M$ are not unifiable.
3. For any ground atom A over P there is an atom B in $\mathcal{I} \cup M$ such that $A = B\sigma$ for some σ .
4. Any literal $L \in \Delta$ is an instance of a literal $K \in M$.
5. Any literal $L \in \Gamma$ is an atom instance of a literal $K \in \mathcal{I}$.
6. The maximal depth of an atom in $\mathcal{I} \cup M$ is at most one larger than the maximal depth of a ground atom in $\Gamma \cup \Delta$.

Lemma 3 (Soundness of Initial State). *The initial state $(\Gamma; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$ is sound.*

Proof. Invariants 1 to 6 given in Definition 2 hold in the initial state:

1. Only $P(x_1, \dots, x_n) \in \mathcal{I} \cup M$ which is linear by definition.
2. Holds trivially, because $\mathcal{I} \cup M$ contains only $P(x_1, \dots, x_n)$.
3. The atom $P(x_1, \dots, x_n) \in \mathcal{I} \cup M$ generalizes all ground atoms over Σ with predicate P .
4. Holds trivially, because $\Delta = \emptyset$.
5. The atom B of $P(x_1, \dots, x_n) \in \mathcal{I}$ generalizes all ground atoms over Σ with predicate P , and any atom(L) $\in \Gamma$ is one of those.
6. Holds trivially, because the maximal depth of $P(x_1, \dots, x_n) \in \mathcal{I}$ is equal to one. □

Lemma 4. (Soundness of \Rightarrow_{mod} Rules). *The rules of \Rightarrow_{mod} preserve state soundness.*

Proof. The proof is carried out by induction over the number of rule applications. By the induction hypothesis, we assume Invariants 1 to 6 given in Definition 2 hold in a state $(\Gamma'; \Delta'; \mathcal{I}'; M')$ and show that after the application of any rule they are still met in the resulting state $(\Gamma; \Delta; \mathcal{I}; M)$.

Rule Refine. The literal $L = P(t_1, \dots, t_n) \in \mathcal{I}'$ is replaced by literals L_i differing from L solely in the position p , which now contains either a constant symbol or a function symbol whose arguments are fresh different variables.

1. The literal L is linear by the induction hypothesis. The variables introduced in the literals L_i are fresh and different, hence all literals L_i are linear, too. The literals in M' are linear by the induction hypothesis and M' remains unaffected. Therefore, all literals in $\mathcal{I} \cup M$ are linear as well.

2. By the induction hypothesis, no two atoms in $\mathcal{I}' \cup M'$ are unifiable. This holds in particular for the atom of L and any other atom in $\mathcal{I}' \cup M'$. The terms $L_i|_p$ are not unifiable by the definition of rule Refine, hence their atoms are not unifiable. Since the literals L_i are instances of L , their atoms are not unifiable with any atom in $(\mathcal{I}' \cup M') \setminus \{L\}$ and thus of $\mathcal{I} \cup M$. Moreover, the atoms $\mathcal{I}' \cup M' \setminus \{L\}$ are by induction hypothesis not unifiable with each other.

3. Let A be any ground atom. By the induction hypothesis, there exists a literal K in $\mathcal{I}' \cup M'$ such that A is an instance of atom(K), i.e., there exists a substitution σ such that $A = \text{atom}(K)\sigma$. If K is not the literal L that is refined, then K is

still in $\mathcal{I} \cup M$ and so the property holds for atom A . If K is the literal L that is refined on position p with $x = L|_p$, then we know that atom A has a term $A|_p = f_i(s_1, \dots, s_{k_i})$ at position p . This means A is also an instance of the literal $L_i = P(t_1, \dots, t_n)\{x \mapsto f_i(y_{i_1}, \dots, y_{k_i})\}$ that was newly added to \mathcal{I} such that $A = \text{atom}(L_i)\sigma_i$, where $\sigma_i = \sigma \cup \{y_{i_1} \mapsto s_1, \dots, y_{k_i} \mapsto s_{k_i}\}$. Hence the property holds for all ground atoms.

4. Both Δ' and M' remain unaffected, and Invariant 4 still holds.

5. The set Γ' remains unaffected by rule Refine, and its atoms are ground atoms over P . By the induction hypothesis, for any atom A in Γ' there exist a literal K in \mathcal{I}' with $\text{atom}(K) = B'$ and a σ such that $B'\sigma = A$. If K is not the literal L that is refined, then K is still in \mathcal{I} and so the property holds for atom A . If K is the literal L that is refined at position p with $x = L|_p$, then we know that atom A has a term $A|_p = f_i(s_1, \dots, s_{k_i})$ at position p . This means A is also an instance of the literal $L_i = P(t_1, \dots, t_n)\{x \mapsto f_i(y_{i_1}, \dots, y_{k_i})\}$ that was newly added to \mathcal{I} such that $A = \text{atom}(L_i)\sigma_i$ where $\sigma_i = \sigma \cup \{y_{i_1} \mapsto s_1, \dots, y_{k_i} \mapsto s_{k_i}\}$. Hence the property holds for all atoms A in Γ' .

6. By the definition of rule Refine, the depth of any literal L_i added to \mathcal{I}' can be at most the depth of $P(t_1, \dots, t_n)$ plus one, due to the introduction of a function symbol at position p . Therefore, the maximal depth of the literals in \mathcal{I}' may increase at most by one. However, the depth of any atom in Γ' which is an instance of $P(t_1, \dots, t_n)$ is at least equal to the one of $P(t_1, \dots, t_n)$. Furthermore, Γ' , Δ' , and M' remain unaltered, and therefore Invariant 6 still holds after the application of rule Refine.

Rule Clean.

1–3,5,6. The sets Γ' and Δ' remain unaltered. The literal $P(t_1, \dots, t_n)$ in \mathcal{I}' is moved from \mathcal{I}' to \mathcal{I} . It remains unaffected, just as any other literal in $\mathcal{I}' \cup M'$, and by the induction hypothesis, Invariants 1–3, 5, and 6 hold.

4. The ground set Δ' remains unchanged, and no literal is removed from M' , therefore Invariant 4 still holds after executing rule Clean.

Rule Solve.

1–3,6. The literal $P(t_1, \dots, t_n)$ is moved from \mathcal{I}' to M , either with positive or negative sign. Its atom remains unaffected, just as any other literal in $\mathcal{I}' \cup M'$, and by the induction hypothesis, Invariants 1–3, and 6 hold.

4. The literals added to Δ are ground instances of $\#P(t_1, \dots, t_n)$ added to M , and Invariant 4 is met after the application of rule Solve.

5. The literals removed from Γ' are ground instances of $\#P(t_1, \dots, t_n)$ and $P(t_1, \dots, t_n)$ is removed from \mathcal{I}' . Therefore, Invariant 5 still holds after applying rule Solve.

6. The removal of $P(t_1, \dots, t_n)$ from \mathcal{I}' and the addition of $\#P(t_1, \dots, t_n)$ to M do not affect the maximal depth of the atoms in $\mathcal{I}' \cup M'$, and $\Gamma' \cup \Delta'$ remains unaffected. Therefore, Invariant 6 still holds after applying rule Solve. \square

Next we show termination and that \Rightarrow_{mod} does not get stuck and always ends in a final state. From now on we only consider sound states.

Lemma 5 (Termination and Runtime). \Rightarrow_{mod} terminates in polynomial time $O(\text{size}(\Gamma)^2)$ with respect to the size of Γ .

Proof. For a state $(\{L_1, \dots, L_n\}; \Delta; \mathcal{I}; M)$ let \mathcal{I}' be a multiset of literals $\{K_1, \dots, K_n\}$ out of literals from \mathcal{I} such that $\text{atom}(L_i) = \text{atom}(K_i)\sigma_i$ for some σ_i . Note that for a given $\{L_1, \dots, L_n\}$ and \mathcal{I} the multiset \mathcal{I}' is unique. This is a result of a sound state and Invariant 2.2. Let

$$\delta(\{L_1, \dots, L_n\}, \{K_1, \dots, K_n\}) = \sum_{1 \leq i \leq n} \text{size}(\sigma_i).$$

Now the measure $(\delta(\{L_1, \dots, L_n\}, \{K_1, \dots, K_n\}), |\mathcal{I}|)$ with $>_{\text{lex}}$ strictly decreases with each rule application: The rules Clean and Solve strictly decrease the number of L_i and/or $|\mathcal{I}|$. For the rule Refine the atom $P(t_1, \dots, t_n)$ has at least two different instances among the L_i and after application of the rule the respective σ_i for all those instances decrease in size by one.

There are at most $\text{size}(\Gamma)$ many applications of Refine possible and for each of these applications at most $\text{size}(\Gamma)$ many applications of Clean or Solve are possible, resulting in the above upper bound. Please recall that the number of symbols in Ω is also bound by $\text{size}(\Gamma)$. \square

Lemma 6 (No Stuck States). *If for a state $(\Gamma; \Delta; \mathcal{I}; M)$ we have $\Gamma \neq \emptyset$ or $\mathcal{I} \neq \emptyset$ then at least one \Rightarrow_{mod} rule is applicable.*

Proof. Suppose $\Gamma \neq \emptyset$ and $L \in \Gamma$. By soundness, Definition 2.5, there exists a literal $K \in \mathcal{I}$ such that $\text{atom}(L)$ is an instance of K . If in addition \mathcal{I} contains a literal H of sign opposite to the one of L where $\text{atom}(H)$ is an instance of K and a minimal variable position p in K such that $\text{atom}(L)|_p$ and $\text{atom}(H)|_p$ are not unifiable, the preconditions of rule Refine are met. If instead all literals $H \in \Gamma$, whose atoms are an instance of K , have the same sign as L , rule Solve can be applied. By Definition 2.5, it can not happen that $\Gamma \neq \emptyset$ and $\mathcal{I} = \emptyset$. Now assume $\Gamma = \emptyset$ and $\mathcal{I} \neq \emptyset$ and let L be a literal in \mathcal{I} . No atom instance of L is contained in Γ , and the preconditions of rule Clean are met. \square

A consequence of Lemma 6 is that \Rightarrow_{mod} always makes progress, i.e., in any non-terminal state, a rule is applicable. Finally, we prove that \Rightarrow_{mod} in fact produces an overall interpretation satisfying the literals from the initial state.

Lemma 7 (All Literals are Considered). *Let $(\Gamma_0; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$ be an initial state. Then for any (possibly non-final) state $(\Gamma; \Delta; \mathcal{I}; M)$ obtained during the execution of \Rightarrow_{mod} on the initial state, it holds that $\Gamma \cup \Delta = \Gamma_0$.*

Proof. In the initial state $(\Gamma_0; \Delta_0; \{P(x_1, \dots, x_n)\}; \emptyset)$, this is obviously the case since $\Delta_0 = \emptyset$. For proving that this property holds throughout the execution of \Rightarrow_{mod} , we assume that it holds in a state $(\Gamma'; \Delta'; \mathcal{I}'; M')$ and show that after applying one rule, it is still met in the resulting state $(\Gamma; \Delta; \mathcal{I}; M)$.

Refine, Clean. Both Γ' and Δ' remain unaltered, hence $\Gamma \cup \Delta = \Gamma_0$.

Solve. Literals are moved from Γ_0 to Δ , hence $\Gamma \cup \Delta = \Gamma_0 \cup \Delta_0 = \Gamma_0$. \square

Lemma 8 (Complete Linear Literal Model). *Let $(\Gamma_0; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$ be an initial state and $(\emptyset; \Delta; \emptyset; M)$ a final state generated by executing \Rightarrow_{mod} on it. Then M is a complete linear literal model of Γ_0 .*

Proof. M is a complete linear literal interpretation by Definition 2.1-3, Lemma 4. By Lemma 7, we have $\Delta = \Gamma_0$. By Definition 2.4, the literals in M generalize all literals in Δ and hence in Γ_0 . This proves that M is a model of Γ_0 . \square

Our rules are not deterministic, and several factors affect the model obtained by running \Rightarrow_{mod} with the same initial state $(\Gamma_0; \emptyset; P(x_1, \dots, x_n); \emptyset)$. If the preconditions of multiple rules are met in a non-final state $(\Gamma; \Delta; \mathcal{I}; M)$, we are free to choose the order in which we execute them. If there are literals $L, K \in \mathcal{I}$ meeting the preconditions of Refine with respect to the same minimal variable position p , either may be chosen. Thus applying \Rightarrow_{mod} to the same trail twice might give us two literal interpretations of different size as shown by an example.

Example 9 (Model Size). Consider the signature $\Sigma = (\{a/0, g/1\}, \{R/2\})$ and $\Gamma_0 = \{L_1, L_2, L_3, L_4, L_5, L_6\}$ where $L_1 = \neg R(a, a)$, $L_2 = R(a, g(a))$, $L_3 = R(g(a), g(g(a)))$, $L_4 = R(a, g(g(a)))$, $L_5 = \neg R(g(a), g(a))$, and $L_6 = \neg R(g(a), a)$. A possible run is shown below. The variables or literals we refine in the next step or apply Solve or Clean to, respectively, are underlined.

$$\begin{aligned}
 0 : & \quad (\Gamma_0; \emptyset; \{R(\underline{x}, y)\}; \emptyset) \\
 1 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma_0; \emptyset; \{R(a, \underline{y}), R(g(z), y)\}; \emptyset) \\
 2 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma_0; \emptyset; \{R(a, \underline{a}), R(a, g(\underline{u})), R(g(z), y)\}; \emptyset) \\
 3 : & \Rightarrow_{\text{mod}}^{\text{Solve}^*} (\Gamma_1; \Delta_1; \{R(g(z), \underline{y})\}; M_1) \\
 4 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma_1; \Delta_1; \{R(g(z), \underline{a}), R(g(z), g(\underline{v}))\}; M_1) \\
 5 : & \Rightarrow_{\text{mod}}^{\text{Solve}} (\Gamma_2; \Delta_2; \{R(g(z), g(\underline{v}))\}; M_2) \\
 6 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma_2; \Delta_2; \{R(g(z), g(\underline{a})), R(g(z), g(g(\underline{w})))\}; M_2) \\
 7 : & \Rightarrow_{\text{mod}}^{\text{Solve}^*} (\Gamma_3; \Delta_3; \emptyset; M_3)
 \end{aligned}$$

The initial state is given by $(\Gamma_0; \emptyset; \{R(x, y)\}; \emptyset)$ (step 0). We choose to refine x at position $p = 1$ in $R(x, y)$, since L_1 and L_3 are instances of its atom with differing signs and $L_1|_1$ and $L_3|_1$ are not unifiable. Rule Refine replaces $R(x, y)$ by $R(a, y)$ and $R(g(z), y)$ (step 1). Similarly, we refine the variable y at position $p = 2$ in $R(a, y)$, since L_1 and L_2 are instances of it having different sign and $L_1|_2$ and $L_2|_2$ are not unifiable (step 2). Then rule Solve can be applied twice, namely to $R(a, a)$ and its negative instance $L_1 \in \Gamma_0$, and to $R(a, g(u))$ and its positive instances $L_2, L_4 \in \Gamma_0$. We obtain $\Gamma_1 = \Gamma_0 \setminus \{L_1, L_2, L_4\}$, $\Delta_1 = \{L_1, L_2, L_4\}$, and $M_1 = \{\neg R(a, a), R(a, g(u))\}$ (step 3). Next, the variable y at position $p = 2$ in $R(g(z), y)$ is refined, since L_3 and L_5 are instances of it having opposite sign and their subterms at position 2 are not unifiable. The literal $R(g(z), y)$ is replaced by $R(g(z), a)$ and $R(g(z), g(v))$ (step 4). Since Γ_1 contains only a positive instance of $R(g(z), a)$, namely L_6 , rule Solve is applied resulting in $\Gamma_2 = \Gamma_1 \setminus \{L_6\}$, $\Delta_2 = \Delta_1 \cup \{L_6\}$, and $M_2 = M_1 \cup \{\neg R(g(z), a)\}$ (step 5). The trail Γ_2 contains instances L_3 and L_5 of $R(g(z), g(v))$ with different sign. Variable v at position 21 is chosen for refinement, since $L_3|_{21}$ and $L_5|_{21}$ are not unifiable,

and $R(g(z), g(v))$ is replaced by $R(g(z), g(a))$ and $R(g(z), g(g(w)))$ (step 6). Now Γ_2 contains only a positive instance of $R(g(z), g(a))$ and a negative one of $R(g(z), g(g(w)))$, and rule Solve is applicable. This gives us $\Gamma_3 = \Gamma_2 \setminus \{L_3, L_5\} = \emptyset$, $\Delta_3 = \Delta_2 \cup \{L_3, L_5\} = \Gamma_0$, and $M_3 = M_2 \cup \{\neg R(g(z), g(a)), R(g(z), g(g(w)))\}$ with 5 literals (step 7).

The choice of the variable to be refined in step 1 is not deterministic, and the following steps might lead to a different model. A different run for $\Gamma'_0 = \Gamma_0$ could be as follows:

$$\begin{aligned}
0 : & (\Gamma'_0; \emptyset; \{R(x, y)\}; \emptyset) \\
1 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma'_0; \emptyset; \{R(x, a), R(x, g(z))\}; \emptyset) \\
2 : & \Rightarrow_{\text{mod}}^{\text{Solve}} (\Gamma'_1; \Delta'_1; \{\overline{R(x, g(z))}\}; M'_1) \\
3 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma'_1; \Delta'_1; \{R(a, g(z)), R(g(u), g(z))\}; M'_1) \\
4 : & \Rightarrow_{\text{mod}}^{\text{Solve}} (\Gamma'_2; \Delta'_2; \{\overline{R(g(u), g(z))}\}; M'_2) \\
5 : & \Rightarrow_{\text{mod}}^{\text{Refine}} (\Gamma'_2; \Delta'_2; \{\overline{R(g(u), g(a))}, \overline{R(g(u), g(g(v)))}\}; M'_2) \\
6 : & \Rightarrow_{\text{mod}}^{\text{Solve}^*} (\Gamma'_3; \Delta'_3; \emptyset; M'_3)
\end{aligned}$$

The first refinement step involves $p = 2$, $y = R(x, y)|_2$ motivated by L_1 and L_2 (step 1). Now we can execute Solve on $R(x, y)$ since it has only negative instances on Γ'_0 , which are L_1 and L_6 obtaining $\Gamma'_1 = \Gamma_0 \setminus \{L_1, L_6\}$, $\Delta'_1 = \Delta_0 \cup \{L_1, L_6\}$, and $M'_1 = \{\neg R(x, a)\}$ (step 2). The variable x at position 1 of $R(x, g(z))$ is refined, since L_2 and L_5 are a positive and negative instance, respectively, of $R(g(u), g(z))$ and their subterms at position 1 are not unifiable, by replacing $R(x, g(z))$ by $R(a, g(z))$ and $R(g(u), g(z))$ (step 3). Now rule Solve can be executed on $R(a, g(z))$, which generalizes L_2 and L_4 , which are both positive. This results in $\Gamma'_2 = \Gamma'_1 \setminus \{L_2, L_4\}$, $\Delta'_2 = \Delta'_1 \cup \{L_2, L_4\}$, and $M'_2 = M'_1 \cup \{R(a, g(z))\}$ (step 4). Next, a Refine step on z at position 2 1 in $R(g(u), g(z))$ is executed due to L_3 and L_5 , which are instances of $R(g(u), g(z))$ of opposite sign and whose subterms at position 2 1 are not unifiable. The literal $R(g(u), g(z))$ is replaced by $R(g(u), g(a))$ and $R(g(u), g(g(v)))$ (step 5), which have one instance each in Γ'_2 . Rule Solve is applied twice, resulting in $\Gamma'_3 = \Gamma'_2 \setminus \{L_3, L_5\} = \emptyset$, $\Delta'_3 = \Delta'_2 \cup \{L_3, L_5\} = \Gamma'_0$, and $M'_3 = M'_2 \cup \{\neg R(g(u), g(a)), R(g(u), g(g(v)))\}$ with 4 literals.

So M_3 and M'_3 not only differ syntactically, but also contain a different number of literals. Refining x before y led to $\neg R(a, a), \neg R(g(z), a) \in M_3$, whereas refining y before x resulted in $\neg R(x, a) \in M'_3$, which generalizes both $\neg R(a, a)$ and $\neg R(g(z), a)$.

In summary, \Rightarrow_{mod} computes an overall interpretation out of the initial finite set of consistent ground literals in polynomial time. We shortly compare our model representation formalism with the long standing literature, in particular [10,17]. They suggested four postulates which should ideally be met by any model representation formalism:

- **Uniqueness.** Each model representation M specifies a single interpretation over Σ .

- **Atom Test.** There exists a fast procedure to evaluate arbitrary ground atoms over the signature Σ in M .
- **Formula Evaluation.** There exists an algorithm deciding the truth value of an arbitrary formula over Σ in M .
- **Equivalence Test.** There exists an algorithm deciding whether two representations M and M' over Σ describe the same interpretation.

The model M obtained by \Rightarrow_{mod} is a complete linear literal interpretation. Our representation formalism is therefore a special case of an atomic representation (ARM) [10] if we leave out negative literals which are implicit for ARMs. The validity of the four model building postulates has been shown for ARMs [10]. So the models computed by \Rightarrow_{mod} satisfy the four model building postulates. Clause evaluation for our linear literal models M is straightforward: a clause C is valid iff there is no substitution σ such that for each $L \in C$ there is a literal $K \in M$ such that $L\sigma$ and $K\sigma$ are complementary. Recall that this is a consequence of the fact that our literal interpretations are explicit and complete: for any ground atom A over Σ there is a literal K in M such that A is a literal instance of K . The respective procedure for ARMs is more involved [45], whereas in our case established techniques for hyper-resolution apply [35, 47, 56].

Finally, we show consequences out of our model building procedure for non-ground literals and the SCL calculus: if the computed interpretation does not satisfy all clauses, then it can be used to effectively compute a minimal extension to the ground literal restriction of the SCL calculus.

Theorem 10 (Non-ground Guarantees). *Let Γ be a set of consistent ground literals. Let M be a model generated by \Rightarrow_{mod} from Γ . Let L be a (potentially non-ground,) linear literal with $\text{depth}(\text{atom}(L)) = d$. Let $\epsilon = 1$ if L has a position p of depth d (i.e., $|p| = d$) such that $L|_p$ is a constant. Otherwise, $\epsilon = 0$. Let Γ contain all ground instances $L\sigma$ of L (i.e., $L\sigma \in \Gamma$) with $\text{depth}(\text{atom}(L\sigma)) \leq d + \epsilon$. Let Γ contain no ground instance of $\text{comp}(L)$, i.e., for all $K \in \Gamma$ it holds that K is not unifiable with $\text{comp}(L)$. Then $M \models L$.*

Proof. Proof by contradiction. We assume that all our assumptions hold, but that $M \not\models L$. By Definition 2.3 and Lemmas 3 and 4, $M \not\models L$ if there exists a $K \in M$ that is unifiable with $\text{comp}(L)$. Moreover, Γ contains no ground instances of $\text{comp}(L)$ by assumption. We will now prove by induction that we can only reach states $(\Gamma'; \Delta; \mathcal{I}; M)$ where $A \in \mathcal{I}$ is unifiable with $\text{atom}(L)$ if A has depth $\leq d + \epsilon$ and Γ' contains all ground instances $L\sigma$ of L such that $\text{atom}(L\sigma)$ is also a ground instance of A and $\text{depth}(\text{atom}(L\sigma)) \leq d + \epsilon$. (Note that there always exists at least one such ground instance because A has depth $\leq d + \epsilon$.) This property guarantees that Clean can never be applied to an atom $A \in \mathcal{I}$ that is unifiable with $\text{atom}(L)$ and that Solve is only applicable to an atom $A \in \mathcal{I}$ that is unifiable with $\text{atom}(L)$ if there is also an instance $\text{atom}(L\sigma)$ of A in Γ' that ensures that we assign A with the correct polarity. The induction base holds trivially because in the state $(\Gamma; \emptyset; \{P(x_1, \dots, x_n)\}; \emptyset)$ the only atom in \mathcal{I} is $P(x_1, \dots, x_n)$ and it has the minimal depth 1 and Γ contains by assumption all ground instances of L with depth $\leq d + \epsilon$. For the induction step, we assume

that $(\Gamma'; \Delta; \mathcal{I}; M)$ is a sound state that satisfies our property and prove that any direct successor state $(\Gamma''; \Delta'; \mathcal{I}'; M')$ must again satisfy our property. We prove this by case distinction:

1) Clean and Solve only remove elements A from \mathcal{I} and all positive and negative instances of A from Γ'' . This together with Definition 2.2 guarantees that the literals removed from Γ'' do not match with any of the remaining elements of \mathcal{I}' . Therefore, the property still holds.

2) Refine on A , but A is not unifiable with $\text{atom}(L)$. Trivial, because any of the new elements in \mathcal{I}' will also not be unifiable with L .

3) Refine on A , A is unifiable with $\text{atom}(L)$, and $\text{depth}(A) \leq d + \epsilon$, and the position p of the refined variable has depth $|p| < d + \epsilon$. This means by induction that $\Gamma' = \Gamma''$ contains all ground instances $L\sigma$ of L such that $\text{atom}(L\sigma)$ is also a ground instance of A and $\text{depth}(\text{atom}(L\sigma)) \leq d + \epsilon$. Moreover, any new atom $A' \in \mathcal{I}' \setminus \mathcal{I}$ has at most depth $|p| + 1$ so still $\leq d + \epsilon$. And lastly, since A' is an instance of A , Γ' contains all ground instances $L\sigma$ of L such that $\text{atom}(L\sigma)$ is also a ground instance of A' and $\text{depth}(\text{atom}(L\sigma)) \leq d + \epsilon$.

4) Refine on A , A is unifiable with $\text{atom}(L)$, $\text{depth}(A) = d + \epsilon$, and the position p of the refined variable has depth $|p| = d + \epsilon$. Let $A^* = \text{mgu}(A, \text{atom}(L))$ as well as $L^* = A^*$ and $L' = A$ if L is positive or else $L^* = \neg A^*$ and $L' = \neg A$. This means by induction that $\Gamma' = \Gamma''$ contains all ground instances $L^*\sigma$ with $\text{depth}(\text{atom}(L^*\sigma)) \leq d + \epsilon$ and that they all have the same polarity as L . Moreover, any variable in A has either a position q with depth $|q| = d + \epsilon$ or there exist no $(A\tau), (\neg A\tau') \in \Gamma'$ such that $(A\tau)|_q$ and $(\neg A\tau')|_q$ are not unifiable. However, this means we also know that any ground instance $(L^*[x_q]_q)\sigma$ of $(L^*[x_q]_q)$ must be in $\Gamma' = \Gamma''$ if q is the variable position of x_q in A with $|q| < d + \epsilon$. Note that due to linearity of L and A (and assuming disjoint variables) $A^*|_q \neq A|_q$ if and only if there exist q', q'' such that $q = q'q''$, $A|_{q'}$ is the position of a variable $x_{q'}$, $|q'| < d + \epsilon$, $L|_{q'}$ is defined and not a variable, and $A^*|_{q'} = L|_{q'}$. This means that we get L'/A if we replace all positions q in L^*/A^* with $A|_q$ if $A|_q = x_q$ and $|q| < d + \epsilon$. If we use this together with the previous fact for all variable positions $|q| < d + \epsilon$, then we get that any ground instance $L'\sigma$ of L' must be in $\Gamma' = \Gamma''$ and therefore Refine is not applicable. A contradiction.

5) Refine on A , A is unifiable with $\text{atom}(L)$, and $\text{depth}(A) > d + \epsilon$. This case is impossible by induction hypothesis! \square

The preconditions of Theorem 10 may look unrelated to its conclusion at first sight. The first example shows why Γ needs to contain all ground instances $L\sigma$ of L of depth $\text{depth}(L)$. The reason is that Refine may lead to an atom K in \mathcal{I} that is unifiable with $\text{comp}(L)$ but Γ contains no ground instances of $\text{comp}(K)$. In our example this is $P(x, f(y))$. The second example shows why Γ needs to contain all ground instances $L\sigma$ of L of depth $\text{depth}(L) + 1$ if L has a constant at a position p with depth $d = \text{depth}(L)$. The reason is that an application of Refine may lead to an atom K in \mathcal{I} that is unifiable with $\text{comp}(L)$ but has no ground instances of $\text{comp}(K)$. In our example this is $P(f(x), y)$.

Example 11. (1) Let $\Gamma = \{\neg P(a, a), \neg P(f(a), a), P(a, f(a))\}$ with signature $\Sigma = (\{a/0, f/1\}, \{P\})$. Then for the input state $(\Gamma; \emptyset; P(x, y); \emptyset)$ the calculus

returns the model $M = \{\neg P(x, a), P(x, f(y))\}$ because we first need to apply Refine to position 2. Although for $\neg P(f(x), y)$ there is no inconsistent atom in Γ , $M \not\models \neg P(f(x), y)$.

(2) Let $\Gamma = \{\neg P(a, a), \neg P(a, b), \neg P(b, a), P(b, b)\}$ with signature $\Sigma = (\{a/0, b/0, f/1\}, \{P\})$. Then for the input state $(\Gamma; \emptyset; P(x, y); \emptyset)$ the calculus can return the model $M = \{\neg P(a, y), P(f(x), y), \neg P(b, a), P(b, b), P(b, f(y))\}$ if we first apply Refine to position 1. And although for $\neg P(x, a)$ there is no inconsistent atom in Γ , $M \not\models \neg P(x, a)$.

Theorem 12 (Non-ground Guarantees by Clean). *Let Γ be a consistent set of ground literals. Let M be a model generated by \Rightarrow_{mod} from Γ . Let d be the maximal depth of any negative literal in Γ . Let A be a linear atom with $\text{depth}(A) \leq d$. Let Γ contain all ground instances of $A\sigma$ with $\text{depth}(A\sigma) \leq d$. Then $M \models A$.*

Proof. Note that the most general unifier of any two linear literals K, K' has depth $\text{depth}(K \text{ mgu}(K, K')) = \max(\text{depth}(K), \text{depth}(K'))$. Firstly, we show that rule Solve can never add a negative literal $\neg B$ to the model that is unifiable with $\neg A$. In this case, Solve is only applicable if Γ contains a ground instance $\neg B\sigma$ and no ground instance $B\sigma$. The first condition implies $\text{depth}(\neg B) \leq d$ and if B and A are unifiable $A \text{ mgu}(A, B)$ has depth $\leq d$. However, since Γ contains all ground instances of A with depth $\leq d$ this also means Γ contains all ground instances of $A \text{ mgu}(A, B)$ with depth $\leq d$. This means that our assumptions guarantee that the second condition for Solve is not satisfied if $\neg B$ is unifiable with $\neg A$. So Solve will not add a literal $\neg B$ that is unifiable with $\neg A$. Secondly, in addition to Solve, only Clean adds literals to M . All literals added by Clean are atoms, so they cannot unify with $\neg A$. Hence, $M \models L$. \square

Lemma 13 (Lower Bound for SCL Refutations). *Let Γ be the ground partial model of an SCL stuck state for the input clause set N and bounded by \preceq and β . This means in particular that (i) every literal $L \in \Gamma$ is ground and bounded by \preceq and β (i.e., $L \preceq \beta$), (ii) every ground atom $A \preceq \beta$ is defined by Γ (i.e., $A \in \Gamma$ or $\neg A \in \Gamma$), and (iii) for every clause $C \in N$ and every grounding σ of C either $\Gamma \models C\sigma$ or there exists a literal $L \in C\sigma$ such that $L \not\preceq \beta$. Let M be a complete interpretation (i.e., for every ground atom A , $M \models A$ or $M \models \neg A$) that models Γ (i.e., $M \models \Gamma$) but not the clause set N (i.e., $M \not\models N$). Let β' be a smallest ground literal according to \preceq such that there exists a clause $C \in N$, a grounding τ , where $L \preceq \beta'$ holds for any literal $L \in C\tau$, and $M \not\models C\tau$. Then there exists no $\beta^* \prec \beta'$ such that an SCL run on N and bounded by \prec and β^* finds a refutation.*

Proof. The assumptions for β' and the completeness of M imply that $M \models C\sigma$ for all clauses $C \in N$ and all groundings σ , where $L \preceq \beta^*$ for any literal $L \in C\sigma$. This means one valid SCL run for N , \preceq , and β^* can simply decide all ground atoms $A \preceq \beta^*$ according to M , i.e., according to whether $M \models A$ or $M \models \neg A$, without encountering any conflicts and ending in a stuck state with a set Γ' such that $\Gamma' \models \text{gnd}^{\preceq \beta^*}(N)$, where the function $\text{gnd}^{\preceq \beta^*}$ computes the set of all

ground instances of a clause set where the grounding is restricted to produce literals L with $L \preceq \beta^*$. The existence of this stuck state proves that $\text{gnd}^{\preceq \beta^*} N$ is satisfiable and that there exists no refutation for it. Hence, no SCL run for N , \preceq , and β^* can find a refutation. \square

5 Conclusion and Future Work

Explicit model building is always a compromise between the expressivity of the used language, its computational properties and the effort to actually compute the model. Satisfiability of first-order logic clause sets is not even semi-decidable, so there cannot be a general solution. In the context of SCL, efficient model building and efficient clause evaluation are important aspects and our quite simple model building language, namely complete linear literal interpretations, nicely serves these two purposes. Still there may be room for improvement. For example, the three clauses

$$\begin{array}{l} \neg R(x, x) \qquad R(x, g(x)) \\ \neg R(x, y) \vee \neg R(y, z) \vee R(x, z) \end{array}$$

do not have a finite model. Linear literal interpretations have the finite model property so there cannot be a finite representation of a model within this language. It needs a more expressive language. For example, assuming an additional constant a and a bound $\beta = R(g(a), g(g(a)))$ a partial model computed by SCL would be

$$[\neg R(a, a), R(a, g(a)), R(g(a), g(g(a))), R(a, g(g(a))), \neg R(g(a), g(a)), \neg R(g(a), a)^1]$$

The respective overall model could be represented by the linear Horn clause set

$$\begin{array}{ll} \neg R(a, a) & R(a, g(a)) \\ \neg R(x, y) \rightarrow \neg R(g(x), y) & R(x, y) \rightarrow R(x, g(y)) \\ \neg R(x, y) \rightarrow \neg R(g(x), g(y)) & R(x, y) \rightarrow R(g(x), g(y)) \end{array}$$

or by terms with exponents and constraints [4, 13]

$$i \geq j \parallel \neg R(g^i(a), g^j(a)) \quad i < j \parallel R(g^i(a), g^j(a)).$$

However, it is an open question how such representations can be actually computed out of a set of ground literals and how they can be used to efficiently test validity of clauses.

The rule Clean may actually add the respective literal wither positively or negatively to M . In practice, such literals could be marked in M . Then in case of starting from an SCL stuck trail where M is not a model for the clause set, a small but useful extension is to check whether flipping the sign of some of these literals turn M into a model.

In summary, we have presented an algorithm that computes in polynomial time out of a finite consistent set of ground literals Γ a complete linear literal

interpretation M such that $M \models \Gamma$. Furthermore, M can be effectively used to evaluate clauses and to determine a minimal extension to the ground literal restriction β out of an SCL stuck state.

Acknowledgements. We thank our anonymous reviewers for their constructive comments.

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Logic* **10**(1), 4:1–4:51 (2009)
2. Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition with simplification as a decision procedure for the monadic class with equality. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) *KGC 1993. LNCS*, vol. 713, pp. 83–96. Springer, Heidelberg (1993). <https://doi.org/10.1007/BFb0022557>
3. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. *J. Appl. Log.* **7**(1), 58–74 (2009)
4. Bensaid, H., Peltier, N.: A complete superposition calculus for primal grammars. *J. Autom. Reason.* **53**(4), 317–350 (2014)
5. Bromberger, M., Desharnais, M., Weidenbach, C.: An Isabelle/HOL formalization of the SCL(FOL) calculus. In: Pientka, B., Tinelli, C. (eds.) *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction. LNCS*, vol. 14132, pp. 116–133. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_7
6. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the Bernays-Schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) *VMCAI 2021. LNCS*, vol. 12597, pp. 511–533. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_23
7. Bromberger, M., Schwarz, S., Weidenbach, C.: Exploring partial models with SCL. In: Konev, B., Schon, C., Steen, A. (eds.) *Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022)*, Haifa, Israel, 11 - 12 August 2022. *CEUR Workshop Proceedings*, vol. 3201 (2022)
8. Bromberger, M., Schwarz, S., Weidenbach, C.: Exploring partial models with SCL. In: Piskac, R., Voronkov, A. (eds.) *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPIc Series in Computing*, vol. 94, pp. 48–72. EasyChair (2023). <https://doi.org/10.29007/8br1>
9. Bromberger, M., Schwarz, S., Weidenbach, C.: SCL(FOL) revisited (2023). <https://doi.org/10.48550/ARXIV.2302.05954>, <https://arxiv.org/abs/2302.05954>
10. Caferra, R., Leitsch, A., Peltier, N.: *Automated Model Building, Applied Logic Series*, vol. 31. Kluwer (2004)
11. Cantone, D., Cutello, V.: A decidable fragment of the elementary theory of relations and some applications. In: Watanabe, S., Nagata, M. (eds.) *Symbolic and Algebraic Computation, Proceedings of the International Symposium, Tokyo, Japan*, pp. 24–29. ACM Press (August 1990)
12. Claessen, K., Soerensson, N.: New techniques that improve MACE-style finite model finding. In: *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications* (2003)

13. Comon, H.: On unification of terms with integer exponents. *Math. Syst. Theory* **28**(1), 67–88 (1995)
14. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Nieuwenhuis, R. (ed.) *RTA 2003*. LNCS, vol. 2706, pp. 148–164. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44881-0_12
15. Fermüller, C.: A resolution variant deciding some classes of clause sets. In: Börger, E., Kleine Büning, H., Richter, M.M., Schönfeld, W. (eds.) *CSL 1990*. LNCS, vol. 533, pp. 128–144. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54487-9_56
16. Fermüller, C.G., Leitsch, A.: Model building by resolution. In: Börger, E., Jäger, G., Kleine Büning, H., Martini, S., Richter, M.M. (eds.) *CSL 1992*. LNCS, vol. 702, pp. 134–148. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56992-8_10
17. Fermüller, C.G., Leitsch, A.: Hyperresolution and automated model building. *J. Log. Comput.* **6**(2), 173–203 (1996)
18. Fermüller, C.G., Leitsch, A., Hustadt, U., Tamet, T.: Resolution decision procedures. In: Robinson, A., Voronkov, A. (eds.) *Handbook of Automated Reasoning*, vol. II, chap. 25, pp. 1791–1849. Elsevier (2001)
19. Fermüller, C., Leitsch, A., Tammet, T., Zamov, N. (eds.): *Resolution Methods for the Decision Problem*. LNCS, vol. 679. Springer, Heidelberg (1993). <https://doi.org/10.1007/3-540-56732-1>
20. Fermüller, C.G., Pichler, R.: Model representation over finite and infinite signatures. *J. Log. Comput.* **17**(3), 453–477 (2007)
21. Fiori, A., Weidenbach, C.: SCL clause learning from simple models. In: Fontaine, P. (ed.) *CADE 2019*. LNCS (LNAI), vol. 11716, pp. 233–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_14
22. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: *LICS*, pp. 295–304 (1999)
23. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009*. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
24. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. *AI Commun.* **24**(2), 195–212 (2011)
25. Georgieva, L., Hustadt, U., Schmidt, R.A.: A new clausal class decidable by hyperresolution. In: Voronkov, A. (ed.) *CADE 2002*. LNCS (LNAI), vol. 2392, pp. 260–274. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45620-1_21
26. Goubault-Larrecq, J.: Deciding \mathcal{H}_1 by resolution. In: *Information Processing Letters*, pp. 401–408 (2005)
27. Hillenbrand, T., Weidenbach, C.: Superposition for bounded domains. In: Bonacina, M.P., Stickel, M.E. (eds.) *Automated Reasoning and Mathematics*. LNCS (LNAI), vol. 7788, pp. 68–100. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36675-8_4
28. Horbach, M., Weidenbach, C.: Decidability results for saturation-based model building. In: Schmidt, R.A. (ed.) *CADE 2009*. LNCS (LNAI), vol. 5663, pp. 404–420. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_30
29. Hustadt, U., Schmidt, R.A.: On evaluating decision procedures for modal logics. In: *Proceedings of 15th International Joint Conference on Artificial Intelligence, IJCAI-97*, pp. 202–207 (1997)
30. Hustadt, U., Schmidt, R.A., Georgieva, L.: A survey of decidable first-order fragments and description logics. *J. Relational Methods Comput. Sci.* **1**, 251–276 (2004)

31. Jacquemard, F., Meyer, C., Weidenbach, C.: Unification in extensions of shallow equational theories. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 76–90. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0052362>
32. Janota, M., Suda, M.: Towards smarter MACE-style model finders. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16–21 November 2018. EPiC Series in Computing, vol. 57, pp. 454–470. EasyChair (2018)
33. Joyner, W.H., Jr.: Resolution strategies as decision procedures. *J. ACM* **23**(3), 398–417 (1976)
34. Jr., R.J.B., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Kuipers, B., Webber, B.L. (eds.) Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, 27–31 July 1997, Providence, Rhode Island, USA, pp. 203–208 (1997)
35. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
36. Lamotte-Schubert, M., Weidenbach, C.: BDI: a new decidable clause class. *J. Log. Comput.* **27**(2), 441–468 (2017)
37. Leidinger, H., Weidenbach, C.: SCL(EQ): SCL for first-order logic with equality. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022. LNCS, vol. 13385, pp. 228–247. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_14
38. Leitsch, A.: Deciding Horn classes by hyperresolution. In: Börger, E., Büning, H.K., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 225–241. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52753-2_42
39. Lynch, C.: Schematic saturation for decision and unification problems. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 427–441. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45085-6_37
40. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 133–182. IOS Press (2021)
41. McCune, W.: Mace4 reference manual and guide. *CoRR cs.SC/0310055* (2003)
42. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference 2001. Proceedings, pp. 530–535. ACM (2001)
43. Nieuwenhuis, R.: Basic paramodulation and decidable theories (extended abstract). In: Proceedings 11th IEEE Symposium on Logic in Computer Science, LICS 1996, pp. 473–482. IEEE Computer Society Press (1996)
44. de Nivelle, H., de Rijke, M.: Deciding the guarded fragments by resolution. *J. Symb. Comput.* **35**(1), 21–58 (2003)
45. Pichler, R.: Algorithms on atomic representations of herbrand models. In: Dix, J., del Cerro, L.F., Furbach, U. (eds.) JELIA 1998. LNCS (LNAI), vol. 1489, pp. 199–215. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49545-2_14
46. Schmidt, R.A., Hustadt, U.: First-order resolution methods for modal logics. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 345–391. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_15

47. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
48. Shumsky, O., Wilkerson, R.W., McCune, W., Erçal, F.: Direct finite first-order model generation with negative constraint propagation heuristic. In: Bryant, B.R., Carroll, J.H., Oppenheim, D., Hightower, J., George, K.M. (eds.) Proceedings of the 1997 ACM symposium on Applied Computing, SAC 1997, San Jose, CA, USA, 28 February - 1 March, pp. 25–29. ACM (1997)
49. Silva, J.P.M., Sakallah, K.A.: Grasp - a new search algorithm for satisfiability. In: International Conference on Computer Aided Design, ICCAD, pp. 220–227. IEEE Computer Society Press (1996)
50. Slaney, J.: FINDER: finite domain enumerator system description. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 798–801. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58156-1_63
51. Slaney, J.K., Surendonk, T.: Combining finite model generation with theorem proving: Problems and prospects. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems, First International Workshop FroCoS 1996, Munich, Germany, 26–29 March 1996, Proceedings. Applied Logic Series, vol. 3, pp. 141–155. Kluwer Academic Publishers (1996)
52. Sturm, T., Voigt, M., Weidenbach, C.: Deciding first-order satisfiability when universal and existential variables are separated. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, USA, 5–8 July 2016, pp. 86–95. ACM (2016)
53. Suda, M., Weidenbach, C., Wischniewski, P.: On the saturation of YAGO. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 441–456. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_38
54. Teucke, A., Weidenbach, C.: Decidability of the monadic shallow linear first-order fragment with straight dismatching constraints. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 202–219. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_13
55. Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: CADE 1999. LNCS (LNAI), vol. 1632, pp. 314–328. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_29
56. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischniewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10


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

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





Synthesis of Recursive Programs in Saturation

Petra Hozzová¹ , Daneshvar Amrollahi² , Márton Hajdu¹ ,
Laura Kovács¹ , Andrei Voronkov^{1,3,4}, and Eva Maria Wagner¹

¹ TU Wien, Vienna, Austria

`petra.hozzova@tuwien.ac.at`

² Stanford University, Stanford, USA

³ University of Manchester, Manchester, UK

⁴ EasyChair, Manchester, UK

Abstract. We turn saturation-based theorem proving into an automated framework for recursive program synthesis. We introduce magic axioms as valid induction axioms and use them together with answer literals in saturation. We introduce new inference rules for induction in saturation and use answer literals to synthesize recursive functions from these proof steps. Our proof-of-concept implementation in the VAMPIRE theorem prover constructs recursive functions over algebraic data types, while proving inductive properties over these types.

Keywords: Program Synthesis · Saturation · Superposition · Induction · Recursion · Theorem Proving

1 Introduction

Program synthesis is the task of constructing a program P satisfying a given specification F , ensuring that P is correct by design [20]. In this paper we work with a functional specification F of the input-output relation of a program P , where F is given as a $\forall\exists$ formula in first-order logic [1, 20]. Validity of a specification formula F ensures that for every input value there exists an output value satisfying F , and therefore there is a function which for every input value gives such an output value. Our goal is to *automatically find a (possibly recursive) program that P computes the output, while preserving F .*

As a complementary approach to formal verification, synthesis is inherently more complex [28]. The complexity is further compounded when we consider reasoning about – and synthesizing – programs using recursion. As a remedy, in this paper we advocate for using automated first-order theorem proving as the reasoning back-end to (recursive) program synthesis.

The work [8] extended the saturation-based first-order theorem proving framework to *saturation-based synthesis framework*. The approach (i) uses saturation-based reasoning to prove that a specification F is valid; (ii) tracks the constructive parts of the proof of F ; (iii) and uses them to synthesize a program P satisfying F . In this paper we complement [8] with support for *recursive program synthesis*. We use recent developments on automating induction in saturation [5, 7, 9, 25] and construct recursive programs based on applications of induction.

$$\begin{array}{ll}
 \text{axioms: } \text{half}(0) \simeq 0 & \text{(H1)} \\
 \text{half}(s(0)) \simeq 0 & \text{(H2)} \\
 \forall x. \text{half}(s(s(x))) \simeq s(\text{half}(x)) & \text{(H3)} \\
 \text{specification: } \forall x \exists y. \text{half}(y) \simeq x & \text{(SD)}
 \end{array}$$

Fig. 1. Axioms of `half` and the $\forall\exists$ -specification for the function computing double.

Illustrative Example. Consider the specification (SD) of Fig. 1, which describes the inverse of the `half` function over natural numbers. Given the axiomatization of `half` in Fig. 1, our approach synthesizes the recursive function `double` as a solution of (SD), defined as:

$$\begin{array}{l}
 \text{double}(0) \simeq 0 \\
 \forall x. \text{double}(s(x)) \simeq s(\text{double}(x))
 \end{array} \tag{1}$$

The framework of [8] fails to synthesize a solution of (SD), as `double` is a recursive program. To the best of our knowledge, there exists no automated approach supporting recursive function synthesis from functional input-output specifications in full first-order logic.

This paper provides a solution in this respect by exploiting the constructive nature of induction. Intuitively, each case of an induction axiom tells us how to construct the desired program for the next recursive step using the program for the previous recursive step. We capture this construction recipe contained in the applications of induction in saturation-based proof search, by utilizing answer literals `ans(r)` [4]. When we use an induction axiom in the proof, we introduce a special term into the answer literal, serving for tracking the program corresponding to the induction axiom. As we prove the cases of the induction axiom, we capture their corresponding programs in the answer literal. Finally, when we derive a clause $C \vee \text{ans}(r)$, where C only contains symbols allowed in a program, we convert the special tracker terms from r into recursive functions, and obtain a program for the initial specification conditioned on $\neg C$.

Contributions. We extend saturation-based first-order theorem proving with recursive program synthesis and bring the following contributions¹:

- We introduce induction axioms, dubbed *magic axioms*, which capture the constructive nature of induction (Sect. 5).
- We convert the magic axioms into formulas used by a saturation-based framework to derive programs using recursion over algebraic data types, i.e., special cases of term algebras. We state necessary requirements for the calculus used in saturation and prove correctness of synthesized programs (Sect. 6).
- We present an extension of the superposition calculus that fulfills our necessary requirement and advocate for superposition reasoning for recursive function synthesis (Sect. 7).

¹ Proofs are given in the extended version [10] of our paper.

- We show that our approach, illustrated initially for natural numbers, naturally extends to programs over arbitrary term algebras (Sect. 8).
- We implement our work in the VAMPIRE prover [16] and survey challenging examples it can synthesize (Sect. 9).

2 Preliminaries

We assume familiarity with standard multi-sorted first-order logic (FOL) with equality. We denote variables by x, y, z, w, u , terms by s, t, r , atoms by A , literals by L , clauses by C, D , formulas by F, G , all possibly with indices. Further, we write σ for Skolem constants. We reserve the symbol \square for the *empty clause* which is logically equivalent to \perp . We write \bar{L} for the literal complementary to L . By \simeq we denote the equality predicate and write $t \not\simeq s$ as a shorthand for $\neg t \simeq s$. We include a conditional term constructor *if*–*then*–*else* in the language, as follows: given a formula F and terms s, t of the same sort, we write *if* F *then* s *else* t to denote the term s if F is true and t otherwise. An *expression* is a term, literal, clause or formula. We write $E[t]$ to denote that the expression E contains the term t . For simplicity, $E[s]$ denotes the expression E where all occurrences of t are replaced by the term s . Formulas with free variables are considered implicitly universally quantified, that is we consider closed formulas.

We use the standard semantics for FOL. For an interpretation function I , we denote the interpretation of a variable x , function symbol f and a predicate symbol p by x^I, f^I, p^I , respectively. We use the notation e^I, F^I also for the interpretation of expressions e and formulas F , respectively. Further, for a variable or a constant a and a value v , we denote by $I\{a \mapsto v\}$ the interpretation function I' such that $a^{I'} = v$ and $b^{I'} = b^I$ for any constant or variable $b \neq a$. We write $F_1, \dots, F_n \vdash G_1, \dots, G_m$ to denote that $F_1 \wedge \dots \wedge F_n \rightarrow G_1 \vee \dots \vee G_m$ is valid, and extend the notation also to validity modulo a theory T .

We recall the standard notion of λ -expressions. Let t be a term and x a variable. Then $\lambda x.t$ denotes a λ -*expression*. For any interpretation I , we define $(\lambda x.t)^I$ as the function f given by $f(v) = t^{I\{x \mapsto v\}}$ for any value v . Moreover, we extend the notation of λ -expressions to also bind constants. Let c be a constant, then $\lambda c.t$ also denotes a λ -*expression*, and its interpretation $(\lambda c.t)^I$ is the function f given by $f(v) = t^{I\{c \mapsto v\}}$ for any value v .

A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two expressions E and E' if $E\theta = E'\theta$; θ is a *most general unifier* (*mgu*) if for every unifier η of E and E' , there exists a substitution μ such that $\eta = \theta\mu$. We denote the mgu of E and E' with $\text{mgu}(E, E')$.

We work with *term algebras* [27], in particular with the special classes of the algebraically defined data types of the natural numbers \mathbb{N} , lists \mathbb{L} , and binary trees \mathbb{BT} .² We denote the sorts of symbols and terms by $:$ (colon), e.g., $f : \tau \rightarrow \alpha$ is a function symbol with domain τ and range α . To emphasize the sort τ of a quantified variable x , we write $\forall x \in \tau$ or $\exists x \in \tau$. For a term algebra sort τ , we denote its constructors with Σ_τ . We fix an arbitrary ordering on the constructors,

² Definitions of these term algebras are in the extended version [10] of this paper.

<p>Superposition (Sup):</p> $\frac{s \simeq t \vee C \quad L[s'] \vee D}{(L[t] \vee C \vee D)\theta}$ <p>where $\theta := \text{mgu}(s, s')$.</p>	<p>Binary resolution (BR):</p> $\frac{A \vee C \quad \neg A' \vee D}{(C \vee D)\theta}$ <p>where $\theta := \text{mgu}(A, A')$.</p>	
<p>Factoring (F): Equality resolution (ER): Equality factoring (EF):</p>		
$\frac{A \vee A' \vee C}{(A \vee C)\theta}$ <p>where $\theta := \text{mgu}(A, A')$.</p>	$\frac{s \not\simeq t \vee C}{C\theta}$ <p>where $\theta := \text{mgu}(s, t)$.</p>	$\frac{s \simeq t \vee s' \simeq t' \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\theta}$ <p>where $\theta := \text{mgu}(s, s')$.</p>

Fig. 2. Simplified superposition calculus Sup.

and denote the i -th constructor in the order by c_i , i.e., $\Sigma_\tau = \{c_1, \dots, c_{|\Sigma_\tau|}\}$. For each c_i , we denote its arity with n_{c_i} . We denote with P_{c_i} the set of argument positions of c_i of the sort τ . We only consider the standard models of term algebras. Programs we synthesize may contain terminating recursive functions $f : \tau \rightarrow \alpha$, where τ is a term algebra type. We define such function f by providing a set of equalities $\{f(c(\bar{x})) \simeq t[\bar{x}, f(x_{j_1}), \dots, f(x_{j_{|P_{c_i}}]})]\}_{c \in \Sigma_\tau}$, where $P_c = \{j_1, \dots, j_{|P_c|}\}$, and t contains no occurrences of f except for the distinguished ones. An example of such a definition is (1).

Saturation and Superposition. Saturation-based proof search implements *proving by refutation* [16]: validity of F is proved by establishing unsatisfiability of $\neg F$. Saturation-based first-order theorem provers work with clauses, rather than with arbitrary formulas. To prove a formula F , the provers negate F and further skolemize it and convert it to clausal normal form (CNF). The CNF of $\neg F$ is denoted by $\text{cnf}(\neg F)$, resulting in a set S of initial clauses. For example, the CNF of the negated and skolemized (SD) is

$$\text{half}(y) \not\simeq \sigma, \tag{2}$$

where σ is a fresh constant used for skolemizing x , and y is implicitly universally quantified. Saturation provers *saturate* S by computing logical consequences of S with respect to a sound inference system \mathcal{I} . Whenever the empty clause \square is derived, the set S of clauses is unsatisfiable and F is valid. We may extend the initial set S with additional clauses C_1, \dots, C_n . If C is derived from this extended set, we say C is derived from S *under additional assumptions* C_1, \dots, C_n .

The *superposition calculus* Sup [22] is the most common inference system for first-order logic with equality. Figure 2 shows a simplified version of Sup. The Sup calculus is *sound* (if \square is derived from F , then F is unsatisfiable) and *refutationally complete* (if F is unsatisfiable, then \square can be derived from it).

3 Recent Developments in Saturation

In this section we summarize recent results relevant to our work.

Program Synthesis in Saturation. Synthesizing (non-recursive) programs in saturation has been initiated in [8]. Here, *computable* and *uncomputable* symbols in the signature are distinguished. Intuitively, computable symbols are those which are allowed to appear in a synthesized program. An expression is *computable* if all symbols it contains are computable. A symbol or an expression is *uncomputable* if it is not computable.

Let A_1, \dots, A_n be closed formulas. Then

$$A_1 \wedge \dots \wedge A_n \rightarrow \forall \bar{x} \exists y. F[\bar{x}, y] \quad (3)$$

is a (*synthesis*) *specification with inputs \bar{x} and output y* .

Consider a computable term $r[\bar{x}]$ such that $A_1 \wedge \dots \wedge A_n \rightarrow \forall \bar{x}. F[\bar{x}, r[\bar{x}]]$ holds. Such an $r[\bar{x}]$ is called a *program* for (3) and a *witness for y in (3)*. If $A_1 \wedge \dots \wedge A_n \rightarrow \forall \bar{x}. (F_1 \wedge \dots \wedge F_n \rightarrow F[\bar{x}, r[\bar{x}]])$ holds for computable formulas F_1, \dots, F_n , then $\langle r[\bar{x}] \bigwedge_{i=1}^n F_i \rangle$ is a *program with conditions F_1, \dots, F_n* for (3).

Saturation-based theorem proving was extended in [8] to a *saturation-based program synthesis framework*. To this end, the clasified negated specification (3) is extended by an *answer literal ans*:

$$A_1 \wedge \dots \wedge A_n \wedge \forall y. (\text{cnf}(\neg F[\bar{\sigma}, y]) \vee \text{ans}(y)) \quad (4)$$

The set of clauses (4) is then saturated. During saturation, upon deriving a clause $C[\bar{\sigma}] \vee \text{ans}(r[\bar{\sigma}])$, where $r[\bar{\sigma}]$ is computable and $C[\bar{\sigma}]$ is computable and does not contain *ans*, the program $\langle r[\bar{x}] \neg C[\bar{x}] \rangle$ with conditions for (3) is recorded and the clause is replaced by $C[\bar{\sigma}]$. This step is called *answer literal removal* within saturation. Once saturation terminates by deriving the empty clause \square , the final program for (3) is constructed by composing the relevant recorded programs with conditions in a nested if–then–else. To support derivation of such clauses $C[\bar{\sigma}] \vee \text{ans}(r[\bar{\sigma}])$ and to ensure that answer literals only have computable arguments, the work of [8] extended the superposition calculus $\mathbb{S}\text{up}$ with new inference rules.

Induction in Saturation. Inductive reasoning has been integrated in saturation [5–7, 9, 25]. The main idea in this body of work is to apply induction by *theory lemma generation*: based on already derived formulas, generate a suitable induction axiom and add it to the search space. To this end, the following induction rule is used:

$$\frac{\bar{L}[t] \vee C}{F \rightarrow \forall x. L[x]} \text{ (Ind)},$$

where $L[t]$ is a ground literal, C is a clause, and $F \rightarrow \forall x. L[x]$ is a valid induction axiom. The conclusion of the Ind rule is clasified, yielding $\text{cnf}(\neg F) \vee L[x]$. This

clause is resolved with the premise $\overline{L}[t] \vee C$ immediately after applying the **Ind** rule and the resulting clause $\text{cnf}(\neg F) \vee C$ is added to the search space.

An example of a valid induction schema is the *structural induction axiom for natural numbers*, where $G[x]$ is any closed formula:

$$(G[0] \wedge \forall y.(G[y] \rightarrow G[s(y)])) \rightarrow \forall x.G[x] \quad (5)$$

When we instantiate the schema with $G[x] := L[x]$, we obtain an axiom that can be used in **Ind**. Since the rule requires $\overline{L}[t]$ to be ground, this instance of **Ind** cannot be applied on (2) and thus is not sufficient for proving (SD) of Fig. 1. To prove formulas with a free variable by induction, we extend **Ind** in Sect. 5.

Note that we can also use a complex formula $G[t]$ in place of the literal $L[t]$ in **Ind**, obtaining a more involved rule, possibly with multiple premises, similarly to a *multi-clause induction rule* [7] or a *induction with arbitrary formulas* [6].

4 Saturation with Induction in Constructive Logic

We first summarize the key challenges our work resolves towards recursive synthesis in saturation, and then present our synthesis approach in Sects. 5–8.

The idea of extracting programs from proofs originates from results in constructive (intuitionistic) logic, starting with Kleene’s realizability [14]. In constructive logic, provability of a formula $\forall \bar{x} \exists y.F[\bar{x}, y]$ implies that there is an algorithm which, given values for \bar{x} , outputs a value for y satisfying $F[\bar{x}, y]$.

We note that the structural induction axiom (5) over natural numbers has computational content, as follows. The program r for $\forall x.G[x]$ can be built from a program r_0 for $G[0]$ and a program r_s for $\forall y.(G[y] \rightarrow G[s(y)])$ as:

$$\begin{aligned} r(0) &\simeq r_0 \\ r(s(y)) &\simeq r_s(r(y)) \end{aligned}$$

For this to be useful, we need to first prove $G[0]$, then prove $\forall y.(G[y] \rightarrow G[s(y)])$, and then use the induction axiom to derive $\forall x.G[x]$. Such an approach towards constructing programs does not however work in saturation-based theorem proving, as saturation does not reduce goals to subgoals [2]. Rather, we add the induction axiom as a theory lemma to the proof search and continue saturation, so we do not have proofs of either $G[0]$ or $\forall y.(G[y] \rightarrow G[s(y)])$. Constructing programs during saturation becomes even more complex when using answer literals, because clauses generated during saturation may contain these literals. For example, if we try to extract a proof of $G[0]$, we may find a proof with an answer literal in it.

To capture the constructive nature of induction and address the above challenges of program synthesis in saturation, we use the following trick. We modify the induction axiom so that it indirectly stores information about the programs for $G[0]$ and $\forall y.(G[y] \rightarrow G[s(y)])$. To do this, instead of adding the induction axiom (5), in Sect. 5 we add what we call a *magic axiom for (5)*, where G has an additional argument for storing the program. In Sect. 6 we further convert our magic axioms into formulas to be used to derive recursive programs in saturation.

5 Induction with Magic Formulas

We first present our approach to *proving* formulas with a free variable by induction. We further extend this approach to *synthesis* in Sect. 6. While our approach works the same way with arbitrary term algebras, for the sake of clarity we first introduce our work for natural numbers and then for general term algebras in Sect. 8.

We use the following *magic axiom*:

$$\left(\exists u_0. G[0, u_0] \wedge \forall y. (\exists w. G[y, w] \rightarrow \exists u_s. G[s(y), u_s]) \right) \rightarrow \forall z. \exists x. G[z, x] \quad (6)$$

Note that all magic axioms are valid, as they are instances of the structural induction axiom (5) with the quantified formula $\exists x. G[t, x]$ in place of $G[t]$. The magicalness of (6) stems from its simple, yet powerful expressiveness: when used in proof search, the variables u_0, u_s in the antecedent capture the programs for the base and step cases, allowing us to construct a program for x in the consequent.

Using axiom (6), we introduce the following variant of the *Ind* rule:

$$\frac{\bar{L}[t, x] \vee C}{\left(\exists u_0. L[0, u_0] \wedge \forall y. (\exists w. L[y, w] \rightarrow \exists u_s. L[s(y), u_s]) \right) \rightarrow \forall z. \exists x. L[z, x]} \quad (\text{MagInd})$$

where the only free variable of $L[t, x]$ is x and C does not contain x .

Example 1. Consider the specification (SD) from Fig. 1. To prove it using superposition, and not yet synthesize the function satisfying (SD), we use the following magic axiom:

$$\left(\exists u_0. \text{half}(u_0) \simeq 0 \wedge \forall y. (\exists w. \text{half}(w) \simeq y \rightarrow \exists u_s. \text{half}(u_s) \simeq s(y)) \right) \rightarrow \forall z. \exists x. \text{half}(x) \simeq z \quad (7)$$

To use (7) in saturation, we clausify it and skolemize the variables y, w, x as $\sigma_y, \sigma_w, \sigma_x(z)$, respectively. The following is a refutational proof of (SD) :

1. $\text{half}(y) \not\approx \sigma$ [negated and skolemized specification (SD)]
2. $\text{half}(u_0) \not\approx 0 \vee \text{half}(\sigma_w) \simeq \sigma_y \vee \text{half}(\sigma_x(z)) \simeq z$ [MagInd with (7)]
3. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\sigma_y) \vee \text{half}(\sigma_x(z)) \simeq z$ [MagInd with (7)]
4. $\text{half}(u_0) \not\approx 0 \vee \text{half}(\sigma_w) \simeq \sigma_y$ [BR 1, 2]
5. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\sigma_y)$ [BR 1, 3]
6. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\text{half}(\sigma_w))$ [Sup 4, 5]
7. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx \text{half}(s(s(\sigma_w)))$ [Sup (H3), 6]
8. $\text{half}(u_0) \not\approx 0$ [ER 7]
9. \square [BR 8, (H2)]

Hence, the magic axiom (6) is sufficient to prove (SD). However, (6) does not suffice to synthesize the program for (SD) from the above proof. Similarly to [8], for synthesis we would use

$$\text{half}(y) \not\approx \sigma \vee \text{ans}(y) \quad (8)$$

instead of clause 1 and obtain a derivation similar to the one above, but with the answer literal $\text{ans}(\sigma_x(\sigma))$. As σ_x is a fresh skolem function, it is uncomputable and not allowed in answer literals. Therefore, simply following the approach of [8] fails to synthesize a recursive program from the proof of (SD). We address the challenge of program construction for the skolem function σ_x in Sect. 6. \square

6 Programs with Primitive Recursion

We now construct recursive programs for proofs using induction over natural numbers (6). As mentioned in Sect. 4, the antecedent of the induction axiom gives us a recipe for constructing the program for the consequent. To capture this dependence of the consequent program x on the antecedent programs u_0, u_s , we convert the magic axiom (6) to its equivalent prenex normal form where $\forall u_0, u_s$ precedes $\exists x$:

$$\exists y, w, \forall u_0, u_s, z. \exists x. \left((G[0, u_0] \wedge (G[y, w] \rightarrow G[s(y), u_s])) \rightarrow G[z, x] \right) \quad (9)$$

The prenex form (9) of the magic axiom (6) allows us to record the dependency on the programs resulting from the base and step cases of induction. For that, we introduce a recursive operator to be used for constructing programs.

Definition 1 (Primitive Recursion Operator). Let $f_1 : \alpha$, and $f_2 : \mathbb{N} \times \alpha \rightarrow \alpha$. The *primitive recursion operator* R for natural numbers and α is:

$$\begin{aligned} R(f_1, f_2)(0) &\simeq f_1 \\ R(f_1, f_2)(s(y)) &\simeq f_2(y, R(f_1, f_2)(y)) \end{aligned}$$

Lemma 2 (Recursive Witness). The expression $R(u_0, \lambda y, w. u_s)(z)$ is a witness for the variable x in (9).

Lemma 2 ensures that we can construct a program for the consequent of the magic axiom given programs for the base case and the step case. We next integrate this construction into our synthesis framework using answer literals. For that we take a close look at the skolemization of induction axiom (9), and define skolem symbols, denoted via rec , for the variable x , capturing the recursive program.

Definition 3 (rec-Symbols). Consider formulas $G[t, x]$ with a single free variable $x : \alpha$ containing a term $t : \mathbb{N}$. For each such formula we introduce a distinct computable function symbol $\text{rec}_{G[t, x]} : \alpha \times \alpha \times \mathbb{N} \rightarrow \alpha$. We will refer to such symbols $\text{rec}_{G[t, x]}$ as *rec-symbols*. When the formula $G[t, x]$ is clear from the context or unimportant for the context, we will simply write rec instead of $\text{rec}_{G[t, x]}$.

A term with a rec -symbol as the top-level functor is called a *rec-term*.

Definition 4 (Magic Formula). *The magic formula for $G[t, x]$ is:*

$$\begin{aligned} &\forall u_0, u_s, z. \\ &\left((G[0, u_0] \wedge (G[\sigma_y, \sigma_w] \rightarrow G[s(\sigma_y), u_s])) \rightarrow G[z, \text{rec}_{G[t, x]}(u_0, u_s, z)] \right) \quad (10) \end{aligned}$$

It is easy to see that magic formula (10) is obtained by skolemizing the prenex normal form of magic axiom (9), where we replace the variables y, w by fresh constants σ_y, σ_w , and the variable x by a fresh $\text{rec}_{G[t,x]}$ -symbol. The constants σ_y, σ_w introduced in (10) are said to be *associated with the $\text{rec}_{G[t,x]}$ -term*. An occurrence of any skolem constant σ_y, σ_w is considered computable if it is an occurrence in the second argument of a $\text{rec}_{G[t,x]}$ -term which it is associated with.

We introduce additional requirements for reasoning with rec -terms to ensure that they always represent the recursive function to be synthesized.

Definition 5 (rec-Compliance). An inference system \mathcal{I} is *rec-compliant* if:

1. \mathcal{I} only introduces rec -terms in the instances of the magic formula (10),
2. \mathcal{I} does not introduce uncomputable symbols into arguments of rec -terms in clauses it derives.

Using a rec -compliant inference system \mathcal{I} , we derive clauses containing rec -terms. These terms correspond to functions constructed using the operator R .

Definition 6 (Recursive Function Term). Let σ_y, σ_w be associated with $\text{rec}(s_1, s_2, t)$. Then we call the term $R(s_1, \lambda\sigma_y, \sigma_w.s_2)(t)$ the *recursive function term corresponding to $\text{rec}(s_1, s_2, t)$* .

For a term r , we denote by r^R the expression obtained from r by iteratively replacing all rec -terms by their corresponding recursive function terms, starting from the innermost ones. Similarly, formula F^R denotes the formula F in which we replace all rec -terms by their corresponding recursive function terms.

Lemma 7 (Recursive Witness for Magic Formulas). Consider the formula obtained from (10) by replacing $\text{rec}_{G[t,x]}(u_0, u_s, z)$ by its corresponding recursive function term $R(u_0, \lambda\sigma_y, \sigma_w.u_s)(z)$:

$$\forall u_0, u_s, z. \left((G[0, u_0] \wedge (G[\sigma_y, \sigma_w] \rightarrow G[s(\sigma_y), u_s])) \rightarrow G[z, R(u_0, \lambda\sigma_y, \sigma_w.u_s)(z)] \right) \quad (11)$$

For every interpretation I , there exists a mapping of skolem constants to values $\{\sigma_y \mapsto v_y, \sigma_w \mapsto v_w\}$ such that I extended by this mapping is a model of (11). As a consequence, formula (11) is satisfiable.

Lemma 7 implies that we can use formula (11) instead of (10) in derivation, while preserving the soundness of the derivations. Soundness of our approach to recursive program synthesis is given next.

Theorem 8 (Semantics of Clauses with Answer Literals and rec -terms). Let C_1, \dots, C_m be clauses and F a formula containing no answer literals and no rec -symbols. Let C be a clause containing no answer literals. Let M_1, \dots, M_l be magic formulas. Assume that using a sound rec -compliant inference system \mathcal{I} , we derive $C \vee \text{ans}(r[\bar{\sigma}])$, where $r[\bar{\sigma}]$ is computable, from the set of clauses

$$\{ C_1, \dots, C_m, M_1, \dots, M_l, \text{cnf}(\neg F[\bar{\sigma}, y] \vee \text{ans}(y)) \}.$$

Then

$$M_1^R, \dots, M_l^R, C_1, \dots, C_m \vdash C^R, F[\bar{\sigma}, r^R[\bar{\sigma}]].$$

That is, under the assumptions $M_1^R, \dots, M_l^R, C_1, \dots, C_m, \neg C^R$, the computable expression $r^R[\bar{x}]$ is a witness for y in $\forall \bar{x} \exists y. F[\bar{x}, y]$.

Based on Theorem 8, if the CNF of A_1, \dots, A_n is among C_1, \dots, C_m , then $r^R[\bar{x}]$ is a witness for y in (3) under the assumptions $M_1^R, \dots, M_l^R, C_1, \dots, C_m, \neg C^R$. The following ensures that we can construct recursive programs with conditions.

Theorem 9 (Recursive Programs). Let $r[\bar{\sigma}]$ be a computable term, and $C[\bar{\sigma}, C_1[\bar{\sigma}], \dots, C_m[\bar{\sigma}]$ be ground computable clauses containing no answer literals and no *rec*-symbols. Assume that using a sound *rec*-compliant inference system \mathcal{I} , we derive the clause $C[\bar{\sigma}] \vee \text{ans}(r[\bar{\sigma}])$ from the CNF of

$$\{ A_1, \dots, A_n, C_1[\bar{\sigma}], \dots, C_m[\bar{\sigma}], M_1, \dots, M_l, \neg F[\bar{\sigma}, y] \vee \text{ans}(y) \}$$

where M_1, \dots, M_l are magic formulas. Then,

$$\langle r^R[\bar{x}] \bigwedge_{j=1}^m C_j[\bar{x}] \wedge \neg C[\bar{x}] \rangle$$

is a program with conditions for (3).

From Theorem 9 we obtain the following key result on program synthesis.

Theorem 10 (Recursive Program Synthesis). Let $P_1[\bar{x}], \dots, P_k[\bar{x}]$, where $P_i[\bar{x}] = \langle r_i^R[\bar{x}] \bigwedge_{j=1}^{i-1} C_j[\bar{x}] \wedge \neg C_i[\bar{x}] \rangle$, be programs with conditions for (3), such that $\bigwedge_{i=1}^n A_i \wedge \bigwedge_{i=1}^k C_i[\bar{x}]$ is unsatisfiable. Then the program $P[\bar{x}]$ defined as

$$\begin{aligned} P[\bar{x}] := & \text{if } \neg C_1[\bar{x}] \text{ then } r_1^R[\bar{x}] \\ & \text{else if } \neg C_2[\bar{x}] \text{ then } r_2^R[\bar{x}] \\ & \dots \\ & \text{else if } \neg C_{k-1}[\bar{x}] \text{ then } r_{k-1}^R[\bar{x}] \\ & \text{else } r_k^R[\bar{x}], \end{aligned}$$

is a program for (3).

7 Recursive Synthesis in Saturation

This section integrates the proving and synthesis steps of Sects. 5–6 into saturation. The crux of our approach is that instead of adding standard induction formulas to the search space, we add magic formulas.

Theorems 9–10 imply that to derive recursive programs, we can use any *rec*-compliant calculus, as long as the calculus supports derivation of clauses

$C \vee \text{ans}(r)$, where r is computable and C is ground, computable, and contains no **rec**-terms nor answer literals. In our work we rely on the extended **Sup** calculus of [8], which we (i) further extend by adding magic formulas alongside standard induction formulas, (ii) make **rec**-compliant by disallowing inferences containing uncomputable **rec**-terms, and (iii) extend by adding more complex rules for introducing conditions into **rec**-terms³. We illustrate these steps by our running example.

Example 2. Using the extended **Sup** calculus, we synthesize the program for the specification of Fig. 1. With the magic formula corresponding to (7),

$$\forall u_0, u_s, z. \left((\text{half}(u_0) \simeq 0 \wedge (\text{half}(\sigma_w) \simeq \sigma_y \rightarrow \text{half}(u_s) \simeq s(\sigma_y))) \rightarrow \text{half}(\text{rec}(u_0, u_s, z)) \simeq z \right), \quad (12)$$

we obtain the following derivation⁴:

1. $\text{half}(y) \not\approx \sigma \vee \text{ans}(y)$ [negated, skolemized specification with answer literal]
2. $\text{half}(u_0) \not\approx 0 \vee \text{half}(\sigma_w) \simeq \sigma_y \vee \text{half}(\sigma_x(z)) \simeq z$ [MagInd with (12)]
3. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\sigma_y) \vee \text{half}(\sigma_x(z)) \simeq z$ [MagInd with (12)]
4. $\text{half}(u_0) \not\approx 0 \vee \text{half}(\sigma_w) \simeq \sigma_y \vee \text{ans}(\text{rec}(u_0, u_s, \sigma))$ [BR 1, 2]
5. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\sigma_y) \vee \text{ans}(\text{rec}(u_0, u_s, \sigma))$ [BR 1, 3]
6. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx s(\text{half}(\sigma_w)) \vee \text{ans}(\text{rec}(u_0, u_s, \sigma))$ [Sup 4, 5]
7. $\text{half}(u_0) \not\approx 0 \vee \text{half}(u_s) \not\approx \text{half}(s(s(\sigma_w))) \vee \text{ans}(\text{rec}(u_0, u_s, \sigma))$ [Sup (H3), 6]
8. $\text{half}(u_0) \not\approx 0 \vee \text{ans}(\text{rec}(u_0, s(s(\sigma_w)), \sigma))$ [ER 7]
9. $\text{ans}(\text{rec}(s(0), s(s(\sigma_w)), \sigma))$ [BR 8, (H2)]
10. \square [answer literal removal 9]

The program recorded in step 10 of the proof is $\text{rec}(s(0), s(s(\sigma_w)), x)^R = R(s(0), \lambda \sigma_w. s(s(\sigma_w)))(x) = f(x)$, where f is defined as:

$$\begin{aligned} f(0) &\simeq s(0) \\ f(s(n)) &\simeq s(f(n)) \end{aligned}$$

Note that while the synthesized program satisfies the specification (SD), it does not match the expected definition of the **double** function from (1). Since the **half** function is rounding down, and the specification does not require the synthesized function to produce even results, the base case was resolved in step 9 with (H2), leading to $f(0) \simeq s(0)$. As a result, we have $f(n) = s(\text{double}(n))$ for any n . \square

Example 2 demonstrates that specification (SD) has multiple solutions and saturation can find a solution different from the intended one. In the next example we modify the specification to have a single solution and synthesize it.

³ The rules can be found in the extended version of this paper [10].

⁴ For the fully detailed derivation, see [10].

Example 3. To synthesize the **double** function, we modify the specification:

$$\text{additional axioms: } \text{even}(0) \tag{E1}$$

$$\neg \text{even}(s(0)) \tag{E2}$$

$$\forall x. (\text{even}(s(s(x))) \leftrightarrow \text{even}(x)) \tag{E3}$$

$$\text{new specification: } \forall x \exists y. (\text{half}(y) \simeq x \wedge \text{even}(y)) \tag{SD'}$$

After negating and skolemizing (SD') and adding the answer literal, we obtain:

$$\text{half}(y) \not\simeq \sigma \vee \neg \text{even}(y) \vee \text{ans}(y) \tag{13}$$

In this case we use the magic axiom for the conjunction $G[t, x] := \text{half}(x) \simeq t \wedge \text{even}(x)$:

$$\begin{aligned} & \left(\exists u_0. (\text{half}(u_0) \simeq 0 \wedge \text{even}(u_0)) \wedge \right. \\ & \quad \left. \forall y. (\exists w. (\text{half}(w) \simeq y \wedge \text{even}(w)) \rightarrow \exists u_s. (\text{half}(u_s) \simeq s(y) \wedge \text{even}(u_s))) \right) \\ & \rightarrow \forall z. \exists x. (\text{half}(x) \simeq z \wedge \text{even}(x)) \end{aligned} \tag{14}$$

We classify the magic formula corresponding to (14), and further resolve it with the premise (13) to obtain:

$$\begin{aligned} & \text{half}(u_0) \not\simeq 0 \vee \neg \text{even}(u_0) \vee \text{half}(\sigma_w) \simeq \sigma_y \vee \text{ans}(\text{rec}(u_0, u_s, \sigma)) \\ & \quad \text{half}(u_0) \not\simeq 0 \vee \neg \text{even}(u_0) \vee \text{even}(\sigma_w) \vee \text{ans}(\text{rec}(u_0, u_s, \sigma)) \\ & \text{half}(u_0) \not\simeq 0 \vee \neg \text{even}(u_0) \vee \text{half}(u_s) \not\simeq s(\sigma_y) \vee \neg \text{even}(u_s) \vee \text{ans}(\text{rec}(u_0, u_s, \sigma)) \end{aligned}$$

The refutation of these clauses follows a similar course to the proof in Example 2. However, u_0 occurring in the literal $\neg \text{even}(u_0)$ forces the proof to use (H1) instead of (H2), and thus the final derived answer literal will be $\text{rec}(0, s(s(\sigma_w)), \sigma)$, corresponding exactly to the function definition of **double** from (1). Note that a derivation of this program in this case requires a saturation prover to apply induction on conjunctions of literals. \square

8 Generalization to Arbitrary Term Algebras

Our approach from Sects. 5–7 generalizes naturally to arbitrary term algebras. This section summarizes the key parts of this generalization.⁵

Let τ be a (possibly polymorphic) term algebra with constructors $\{c_1, \dots, c_n\}$, where we denote the sort of each c_i by $\tau_{i,1} \times \dots \times \tau_{i,n_{c_i}} \rightarrow \tau$,

⁵ We state all definitions, lemmas and theorems in the appendix of the extended version of our paper [10].

and $P_{c_i} = \{j_1, \dots, j_{|P_{c_i}|}\}$ for each $i = 1, \dots, n$. Let α be any sort. The *magic axiom for $G[t, x]$* , where $t : \tau, x : \alpha$, is:

$$\left(\bigwedge_{c \in \Sigma_\tau} \forall_{i=1}^{n_c} y_{c,i}. \left(\bigwedge_{j \in P_c} \exists w_{c,j}. G[y_{c,j}, w_{c,j}] \right) \rightarrow \exists u_c. G[c(\overline{y_c}), u_c] \right) \rightarrow \forall z. \exists x. G[z, x] \quad (15)$$

The corresponding *magic formula* uses the skolem function $\text{rec}_{G[t,x]} : \alpha^{n_c} \times \tau \rightarrow \alpha$:

$$\forall_{c \in \Sigma_\tau} u_c. \forall z. \left(\bigwedge_{c \in \Sigma_\tau} \left(\bigwedge_{j \in P_c} G[\sigma_{y_{c,j}}, \sigma_{w_{c,j}}] \rightarrow G[c(\overline{\sigma_{y_c}}), u_c] \right) \rightarrow G[z, \text{rec}_{G[t,x]}(\overline{u}, z)] \right) \quad (16)$$

Note that each $\sigma_{y_{c,i}}, \sigma_{w_{c,j}}$ introduced in (16) is considered computable only in the i th argument of its associated rec -term. We define the *recursion operator* R for τ and α analogously to Definition 1:

$$\begin{aligned} R(f_1, \dots, f_n)(c_1(\overline{x})) &\simeq f_1(x_1, \dots, x_{n_{c_1}}, R(f_1, \dots, f_n)(x_{j_1}), \dots, R(f_1, \dots, f_n)(x_{j_{|P_{c_1}|}})) \\ &\quad \dots \\ R(f_1, \dots, f_n)(c_n(\overline{x})) &\simeq f_n(x_1, \dots, x_{n_{c_n}}, R(f_1, \dots, f_n)(x_{j_1}), \dots, R(f_1, \dots, f_n)(x_{j_{|P_{c_n}|}})) \end{aligned}$$

where for each i we have $f_i : \tau_{i,1} \times \dots \times \tau_{i,n_{c_i}} \times \alpha^{|P_{c_i}|} \rightarrow \alpha$. Using R , we state an analogue of Lemma 7:

Lemma 11 (Recursive Witness for Magic Formulas Using τ). Consider the formula obtained from (16) by replacing $\text{rec}_{G[t,x]}(\overline{u}, z)$ by its corresponding recursive function term:

$$\begin{aligned} \forall_{c \in \Sigma_\tau} u_c. \forall z. \left(\bigwedge_{c \in \Sigma_\tau} \left(\bigwedge_{j \in P_c} G[\sigma_{y_{c,j}}, \sigma_{w_{c,j}}] \rightarrow G[c(\overline{\sigma_{y_c}}), u_c] \right) \right. \\ \left. \rightarrow G[z, R(\lambda_{i=1}^{n_{c_1}} \sigma_{y_{c_1,i}} \cdot \lambda_{k \in P_{c_1}} \sigma_{w_{c_1,k}} \cdot u_{c_1}, \dots, \lambda_{i=1}^{n_{c_n}} \sigma_{y_{c_n,i}} \cdot \lambda_{k \in P_{c_n}} \sigma_{w_{c_n,k}} \cdot u_{c_n})(z)] \right) \quad (17) \end{aligned}$$

For every interpretation, there exists its extension by some $\{\sigma_{y_{c,i}} \mapsto v_{y,c,i}, \sigma_{w_{c,k}} \mapsto v_{w,c,k}\}_{c \in \Sigma_\tau, i \in \{1, \dots, n_c\}, k \in P_c}$ such that the extension is a model of (17). As a consequence, formula (17) is satisfiable.

Using Lemma 11, we derive the analogues of Theorems 8–10 for an arbitrary term algebra τ . We then employ magic formulas (16) in MagInd when in the premise $L[t, x] \vee C \vee \text{ans}(r[x])$ we have $t : \tau$. We finally note that our synthesis method generalizes also to sorts other than term algebras, as long as the induction axiom used for the sort carries the constructive meaning described in Sect. 4.

9 Implementation and Examples

Implementation. We extended the first-order theorem prover VAMPIRE [16] with a proof-of-concept implementation of our method for recursive program

synthesis in saturation. Our implementation consists of approximately 1,100 lines of C++ code and is available online at <https://github.com/vprover/vampire/tree/synthesis-recursive>.

We implemented the `MagInd` rule as well as a version of `MagInd` using a magic axiom with base case $s(0)$ for natural numbers and `cons(a, nil)` for any a for lists. To support synthesis requiring induction on specifications $\neg F[t, x]$, where $F[t, x]$ is an arbitrary formula with the only free variable x , we use an encoding as follows. We change the specification $\forall \bar{x} \exists y. F[\bar{x}, y]$ to $\forall \bar{x} \exists y. p(\bar{x}, y)$, where p is a fresh uncomputable predicate, and we add an axiom $\forall \bar{x}, y. (p(\bar{x}, y) \leftrightarrow F[\bar{x}, y])$.

Table 1. Synthesis examples using natural numbers \mathbb{N} , lists \mathbb{L} and binary trees \mathbb{BT} . The x -variables in the program and synthesized definitions are the inputs. While our framework synthesizes all these examples, our implementation in VAMPIRE only synthesizes those marked with “✓”. Note that for “Length of 2 concatenated lists” we consider $\#$ to be uncomputable.

Specification	Program	Synthesized definitions	VAMPIRE
Double: $\forall x \in \mathbb{N}. \exists y \in \mathbb{N}.$ ($\text{half}(y) \simeq x \wedge \text{even}(y)$)	$f(x)$	$f(0) \simeq 0$ $f(s(n)) \simeq s(f(n))$	✓
Associativity of addition: $\forall x_1, x_2, x_3 \in \mathbb{N}. \exists y \in \mathbb{N}.$ ($x_1 + x_2$) + $x_3 \simeq x_1 + y$	$f(x_3)$	$f(0) \simeq x_2$ $f(s(n)) \simeq s(f(n))$	✓
Subtraction with condition: $\forall x_1, x_2 \in \mathbb{N}. \exists y \in \mathbb{N}.$ ($x_2 < x_1 \rightarrow x_2 + y \simeq x_1$)	$f(x_2)$	$f(0) \simeq x_1$ $f(s(n)) \simeq p(f(n))$	✓
Floored square root: $\forall x \in \mathbb{N}. \exists y \in \mathbb{N}.$ ($y \cdot y \leq x \wedge x < s(y) \cdot s(y)$)	$f(x)$	$f(0) \simeq 0$ $f(s(n)) \simeq$ if $s(n) \simeq s(f(n)) \cdot s(f(n))$ then $s(f(n))$ else $f(n)$	✗
Floored division: $\forall x_1, x_2 \in \mathbb{N}. \exists y \in \mathbb{N}. (x_2 \not\simeq 0 \rightarrow$ ($y \cdot x_2 \leq x_1 \wedge x_1 < s(y) \cdot x_2$)	$f(x_1)$	$f(0) \simeq 0$ $f(s(n)) \simeq$ if $s(n) \simeq s(f(n)) \cdot x_2$ then $s(f(n))$ else $f(n)$	✗
Length of 2 concatenated lists: $\forall x_1, x_2 \in \mathbb{L}. \exists y \in \mathbb{N}.$ $y \simeq \text{len}(x_1 x_2)$	$f(x_1)$	$f(\text{nil}) \simeq \text{len}(x_2)$ $f(\text{cons}(n, l)) \simeq s(f(l))$	✓
Last element of a list: $\forall x \in \mathbb{L}. \exists y \in \mathbb{N}. (x \not\simeq \text{nil} \rightarrow$ $\exists z \in \mathbb{L}. x \simeq z \text{cons}(y, \text{nil}))$	$f(x)$	$f(\text{cons}(n, \text{nil})) \simeq n$ $l \not\simeq \text{nil} \rightarrow f(\text{cons}(n, l)) \simeq f(l)$	✓
Prefix of a list given its suffix: $\forall x_1, x_2 \in \mathbb{L}. \exists y \in \mathbb{L}.$ ($\text{suff}(x_2, x_1) \rightarrow x_1 \simeq y x_2$)	$f(x_2)$	$f(\text{nil}) \simeq x_1$ $f(\text{cons}(n, l)) \simeq g(f(l))$ $g(\text{cons}(n, \text{nil})) \simeq \text{nil}$ $l \not\simeq \text{nil} \rightarrow g(\text{cons}(n, l)) \simeq \text{cons}(n, g(l))$	✗
Maximum element of a list: $\forall x \in \mathbb{L}. \exists y \in \mathbb{N}. (x \not\simeq \text{nil} \rightarrow$ ($\text{in}(y, x) \wedge \forall k \in \mathbb{N}. (\text{in}(k, x) \rightarrow k \leq y)$)	$f(x)$	$f(\text{cons}(n, \text{nil})) \simeq n$ $l \not\simeq \text{nil} \rightarrow f(\text{cons}(n, l)) \simeq$ if $f(l) < n$ then n else $f(l)$	✗
Maximum element of a tree: $\forall x \in \mathbb{BT}. \exists y \in \mathbb{N}.$ ($\text{in}(y, x) \wedge \forall k \in \mathbb{N}. (\text{in}(k, x) \rightarrow k \leq y)$)	$f(x)$	$f(\text{leaf}(n)) \simeq n$ $f(\text{bt}(l, n, r)) \simeq$ if $f(l) < f(r)$ then if $f(l) < n$ then if $f(r) < n$ then n else $f(r)$ else $f(r)$ else if $f(r) < n$ then if $f(l) < n$ then n else $f(l)$ else $f(l)$	✗

Examples. Our implementation can synthesize the programs for the specifications (SD) and (SD'). We also synthesize further examples over the term algebras⁶ of natural numbers \mathbb{N} , lists \mathbb{L} , and binary trees \mathbb{BT} . We display the specifications alongside the programs synthesized by our framework in Table 1. Our framework synthesizes programs for each of the examples⁷, yet our implementation supports so far only a limited set of magic formulas; therefore, the “VAMPIRE” column of Table 1 lists which examples are solved in practice.

Experimental Comparison. To the best of our knowledge, no other approach in program synthesis supports the setting we consider: functional relational specifications of recursive programs, given in full first-order logic, without user-defined templates. For this reason, we could not compare the practical aspects of our work with other techniques, but overview related works in Sect. 10. In particular, we note that the tools surveyed in overview in Sect. 10 support a more restrictive/decidable logic than the the full first-order setting exploited in our approach. As such, the benchmarks of Table 1 cannot be translated into the input languages of techniques surveyed in Sect. 10.

10 Related Work

Our approach is conceptually different from existing methods in recursive program synthesis, as we are not restricted to decidable logical fragments, nor to user-defined program templates. Our work supports program specifications in full first-order logic (with theories) and does not require syntactic templates for the programs to be synthesized. In the sequel, we only discuss related approaches that support *full automation* in program synthesis, without templates or user guidance.

We extend the recursion-free synthesis framework of [8], while exploiting ideas from deductive synthesis [17, 20, 29] using answer literals [4]. We bring recursive program synthesis into the landscape of saturation-based proving and construct programs from saturation proofs with magic axioms. Unlike our setting, the works of [19, 29] construct recursive programs from proofs by induction, by reducing the program specification to subgoals corresponding to the cases of the induction axiom. Modern first-order theorem provers mostly implement saturation-based proof search, which however does not support a goal-subgoal architecture. Our approach integrates induction directly into saturation and enables automated reasoning with term algebras.

⁶ See the extended version of this paper [10] for term algebra constructors and signatures, and for axiomatization and lemmas for the used predicates and functions.

⁷ We provide the full derivations of the synthesized programs in [10].

Fully automated methods supporting recursive program synthesis include SYNQUID [23], LEON [15], JENNISYS [18], SUSLIK [24], CYPRESS [12], BURST [21], and SYNTREC [11]. Except for BURST and SYNTREC, all these works decompose goals into subgoals. Our work complements these methods, by turning saturation into a recursive synthesis framework over first-order theories. As such, our work also differs from SYNQUID, where term enumeration combined with type checking is used over program specifications within decidable logics. LEON uses recursive schemas corresponding to our recursive operator R , instantiates them by candidate program terms, and checks if they satisfy the specification. Unlike LEON, we support a complete handling of quantifiers via superposition reasoning. JENNISYS uses a verifier to generate input-output examples, which differs from our setting of using inductive formulas as logical specifications. BURST generates programs by composition from existing ones, using quantifier-free fragments of first-order logic. Contrarily to this, we support full first-order logic and induction, without using subgoal proof strategies. Finally, we note that SYNTREC guarantees bounded/relative correctness of the synthesized programs (using syntactic program templates), while our approach proves correctness of the synthesized program without further restrictions.

The syntax-guided synthesis (SyGuS) framework [1] supports specifications for recursive functions and can encode our examples from Sect. 9. However, to the best of our knowledge, SyGuS methods, including the SMT-based approach of [26], do not support recursive synthesis. While the semantics-guided synthesis framework [13] also supports recursive functions, its (to the best of our knowledge) only solvers MESSY [13] and MESSY-ENUM [3] synthesize programs from input-output examples and using grammars, respectively, rather than purely from logical specifications.

11 Conclusions

We extend saturation-based framework to recursive program synthesis by utilizing the constructive nature of induction axioms. We introduce magic axioms as a tracking mechanism and seamlessly integrate these axioms into saturation. We then construct correct recursive programs using answer literals in saturation, as also demonstrated by our proof-of-concept implementation. Extending our work with tailored handling of (more general) magic axioms, and respective superposition inferences, is an interesting line for future work. Devising and implementing further, and potentially more general, synthesis rules and induction schemes is another task for future research, allowing us to further strengthen the practical use of our work.

Acknowledgements. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, pp. 1–25 (2015)
2. Bonacina, M.P.: A Taxonomy of theorem-proving strategies. In: Artificial Intelligence Today: Recent Trends and Developments, pp. 43–84 (1999). https://doi.org/10.1007/3-540-48317-9_3
3. D’Antoni, L., Hu, Q., Kim, J., Reps, T.: Programmable program synthesis. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 84–109. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_4
4. Green, C.: Theorem-proving by resolution as a basis for question-answering systems. *Mach. Intell.* **4**, 183–205 (1969)
5. Hajdu, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with Generalization in Superposition Reasoning. In: CICM, pp. 123–137 (2020) https://doi.org/10.1007/978-3-030-53518-6_8
6. Hajdu, M., Kovács, L., Rawson, M., Voronkov, A.: The Vampire Approach to Induction. In: Practical Aspects of Automated Reasoning (2022)
7. Hajdu, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: FMCAD, pp. 1–10 (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_34
8. Hozzová, P., Kovács, L., Norman, C., Voronkov, A.: Program synthesis in saturation. In: CADE, pp. 307–324 (2023)
9. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: CADE, pp. 361–377 (2021)
10. Hozzová, P., Amrollahi, D., Hajdu, M., Kovács, L., Voronkov, A., Wagner, E.M.: Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair (2024)
11. Inala, J.P., Polikarpova, N., Qiu, X., Lerner, B.S., Solar-Lezama, A.: Synthesis of recursive ADT transformations from reusable templates. In: TACAS, pp. 247–263 (2017). https://doi.org/10.1007/978-3-662-54577-5_14
12. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: PLDI, pp. 944–959 (2021). <https://doi.org/10.1145/3453483.3454087>
13. Kim, J., Hu, Q., D’Antoni, L., Reps, T.: Semantics-guided synthesis. *Proc. ACM Program. Lang.* **5**(POPL), 1–32 (2021). <https://doi.org/10.1145/3434311>
14. Kleene, S.: On the interpretation of intuitionistic number theory. *J. Symb. Log.* **10**, 109–124 (1945)
15. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA, pp. 407–426 (2013). <https://doi.org/10.1145/2509136.2509555>
16. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV, pp. 1–35 (2013)
17. Lee, R.C.T., Waldinger, R.J., Chang, C.L.: An improved program-synthesizing algorithm and its correctness. *Commun. ACM* **4**, 211–217 (1974). <https://doi.org/10.1145/360924.360967>
18. Leino, K.R.M., Milicevic, A.: Program extrapolation with Jennisys. In: OOPSLA, pp. 411–430. OOPSLA ’12 (2012). <https://doi.org/10.1145/2384616.2384646>
19. Manna, Z., Waldinger, R.: Fundamentals of deductive program synthesis. *IEEE Trans. Softw. Eng.* **18**(8), 674–704 (1992). <https://doi.org/10.1109/32.153379>
20. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.* **2**(1), 90–121 (1980). <https://doi.org/10.1145/357084.357090>

21. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution. **6**(POPL), 1–29 (2022). <https://doi.org/10.1145/3498682>
22. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasoning, vol. I, pp. 371–443. Elsevier and MIT Press (2001)
23. Polikarpova, N., Kuraaj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* **51**(6), 522–538 (2016). <https://doi.org/10.1145/2980983.2908093>
24. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. **3**(POPL), 1–30 (2019). <https://doi.org/10.1145/3290385>
25. Reger, G., Voronkov, A.: Induction in Saturation-Based Proof Search. In: CADE, pp. 477–494 (2019)
26. Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C.W., Deters, M.: Refutation-based synthesis in SMT. *Formal Methods Syst. Des.* **55**(2), 73–102 (2019). <https://doi.org/10.1007/S10703-017-0270-2>
27. Rybina, T., Voronkov, A.: A decision procedure for term algebras with queues. *ACM Trans. Comput. Log.* **2**(2), 155–181 (2001). <https://doi.org/10.1145/371316.371494>
28. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010). <https://doi.org/10.1145/1706299.1706337>
29. Tammet, T.: Completeness of resolution for definite answers. *J. Logic Comput.* **5**(4), 449–471 (1995)



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

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





Synthesizing Strongly Equivalent Logic Programs: Beth Definability for Answer Set Programs via Craig Interpolation in First-Order Logic

Jan Heuer  and Christoph Wernhard ^(✉) 

University of Potsdam, Potsdam, Germany
{jan.heuer, christoph.wernhard}@uni-potsdam.de

Abstract. We show a projective Beth definability theorem for logic programs under the stable model semantics: For given programs P and Q and vocabulary V (set of predicates) the existence of a program R in V such that $P \cup R$ and $P \cup Q$ are strongly equivalent can be expressed as a first-order entailment. Moreover, our result is effective: A program R can be constructed from a Craig interpolant for this entailment, using a known first-order encoding for testing strong equivalence, which we apply in reverse to extract programs from formulas. As a further perspective, this allows transforming logic programs via transforming their first-order encodings. In a prototypical implementation, the Craig interpolation is performed by first-order provers based on clausal tableaux or resolution calculi. Our work shows how definability and interpolation, which underlie modern logic-based approaches to advanced tasks in knowledge representation, transfer to answer set programming.

1 Introduction

Answer set programming [3, 35, 50, 57, 60] is one of the major paradigms in knowledge representation. A problem is expressed declaratively as a logic program, a set of rules in the form of implications. An answer set solver returns representations of its *answer sets* or *stable models* [36, 49]. That is, minimal Herbrand models, where models with facts not properly justified in a non-circular way are excluded. Modern answer set solvers such as *clingo* [34] are advanced tools that integrate SAT technology.

Two logic programs can be considered as *equivalent* if and only if they have the same answer sets. However, if two equivalent programs are each combined with some other program, the results are not necessarily equivalent. Thus, it is of much more practical relevance to consider instead a notion of equivalence that guarantees the same answer sets even in combination with other programs: Two logic programs P, Q are *strongly equivalent* [54] if they can be exchanged in the

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 457292495.

© The Author(s) 2024

C. Benz Müller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 172–193, 2024.

https://doi.org/10.1007/978-3-031-63498-7_11

context of any other program R without affecting the answer sets of the overall program. That is, P and Q are strongly equivalent if and only if for all logic programs R it holds that $P \cup R$ and $Q \cup R$ have the same answer sets.

Although it has been known that strong equivalence of logic programs under the stable model semantics can be translated into equivalence of classical logical formulas, e.g. [55], developments in the languages of answer set programming make this an issue of ongoing research [18, 25, 38, 39, 42, 53]. The practical objective is to apply first-order provers to determine the equivalence of two logic programs.

We now consider the situation where only a single program is given and a strongly equivalent one is to be synthesized automatically. For the new program the set of allowed predicates, including to some degree the position within rules in which they are allowed, is restricted to a given vocabulary. Not just “absolute” strong equivalence is of interest, but also strong equivalence with respect to some background knowledge expressed as a logic program. Thus, for given programs P and Q , and vocabulary V we want to find programs R in V such that $P \cup R$ and $Q \cup R$ are strongly equivalent.

Our question has two aspects: characterizing the *existence* of a program R for given P, Q, V and, if one exists, the effective *construction* of such an R . As we will show, existence can be addressed by Beth definability [7, 21] on the basis of Craig interpolation [20] for first-order logic. The construction can then be performed by extracting an interpolant from a proof of the first-order characterization of existence. We realize this practically with the first-order provers **CMProver** [74, 75] and **Prover9** [58] and an interpolation technique for clausal tableaux [76, 77].

To achieve this, we start from a known representation of logic programs in classical first-order logic for the purpose of verifying strong equality. We supplement it with a formal characterization to determine whether an arbitrary given first-order formula represents a logic program, and, if so, to extract a represented logic program from the formula. This novel “reverse translation” also has other potential applications in program transformation.

Beth definability and Craig interpolation play a key role for advanced tasks in other fields of knowledge representation, in particular for query reformulation in databases [5, 6, 59, 66, 72] and description logics as well as ontology-mediated querying [2, 16, 17, 73]. Our work aims to provide for these lines of research the bridge to answer set programming.

Structure of the Paper. After providing in Sect. 2 background material on strong equivalence as well as on interpolation and definability we develop in Sect. 3 our technical results. Our prototypical implementation¹ is then described in Sect. 4. We conclude in Sect. 5 with discussing related work and perspectives.

¹ Available from <http://cs.christophwernhard.com/pie/asp> as free software.

2 Background

2.1 Notation

We map between two formalisms: logic programs and formulas of classical first-order logic without equality (briefly *formulas*). In both formalisms we have *atoms* $p(t_1, \dots, t_n)$, where p is a *predicate* and t_1, \dots, t_n are terms built from *functions*, including *constants*, and *variables*. We assume variable names as case insensitive to take account of the custom to write them uppercase in logic programs and lowercase in first-order logic. Predicates in logic programs are distinct from those in formulas, but with a correspondence: If p is a predicate for use in logic programs, then the two predicates p^0 and p^1 , both with the same arity as p , are for use in formulas. Thus, predicates in formulas are always decorated with a 0 or 1 superscript. To emphasize this, we sometimes speak of *0/1-superscripted formulas*.

A *literal* is an atom or a negated atom. A *clause* is a disjunction of literals, a *clausal formula* is a conjunction of clauses. The empty disjunction is represented by \perp , the empty conjunction by \top . On occasion we write a clause also as an implication.

A subformula occurrence in a formula has *positive (negative) polarity* if it is in the scope of an even (odd) number of possibly implicit negations. A formula is *universal* if occurrences of \forall have only positive polarity and occurrences of \exists have only negative polarity. Semantic entailment and equivalence of formulas are expressed by \models and \equiv .

Let F be a formula. $\mathcal{F}un(F)$ is the set of functions occurring in it, including constants, and $\mathcal{P}red(F)$ is the set of predicates occurring in it. $\mathcal{P}red^\pm(F)$ is the set of pairs $\langle pol, p \rangle$, where p is a predicate and $pol \in \{+, -\}$, such that an atom with predicate p occurs in F with the polarity indicated by pol . We write $\langle +, p \rangle$ and $\langle -, p \rangle$ succinctly as $+p$ and $-p$. To map from the predicates occurring in a formula to predicates of logic programs we define $\mathcal{P}red^{LP}(F) \stackrel{\text{def}}{=} \{p \mid p^i \in \mathcal{P}red(F), i \in \{0, 1\}\}$. For logic programs P , we define $\mathcal{F}un(P)$ and $\mathcal{P}red(P)$ analogously as for formulas.

2.2 Strong Equivalence as First-Order Equivalence

We consider *disjunctive logic programs with negation in the head* [48, Sect. 5], which provide a normal form for answer set programs [12]. A *logic program* is a set of *rules* of the form

$$A_1; \dots; A_k; \mathbf{not} A_{k+1}; \dots; \mathbf{not} A_l \leftarrow A_{l+1}, \dots, A_m, \mathbf{not} A_{m+1}, \dots, \mathbf{not} A_n,$$

where A_1, \dots, A_n are atoms, $0 \leq k \leq l \leq m \leq n$. Recall from Sect. 2.1 that an atom can have argument terms with functions and variables. The *positive/negative head* of a rule are the atoms A_1, \dots, A_k and A_{k+1}, \dots, A_l respectively. Analogously the *positive/negative body* of a rule are A_{l+1}, \dots, A_m and A_{m+1}, \dots, A_n respectively. Answer sets with respect to the stable model semantics for this class of programs are for example defined in [32].

Next we review the definition of the translation γ used to express strong equivalence of two logic programs in classical first-order logic.² It makes use of the fact that strong equivalence can be expressed in the intermediate logic of here-and-there [54], which in turn can be mapped to classical logic [62]. For details and proofs we refer to [25, 38, 39]. Similar results appeared in [28, 55, 62, 63]. As stated in Sect. 2.1 we assume for each program predicate p two dedicated formula predicates p^0 and p^1 with the same arity. If A is an atom with predicate p , then A^0 is A with p^0 instead of p , and A^1 is A with p^1 instead of p .

Definition 1. For a rule

$$R = A_1; \dots; A_k; \mathbf{not} A_{k+1}; \dots; \mathbf{not} A_l \leftarrow A_{l+1}, \dots, A_m, \mathbf{not} A_{m+1}, \dots, \mathbf{not} A_n.$$

with variables \mathbf{x} define the first-order formulas $\gamma^0(R)$ and $\gamma^1(R)$ as

$$\begin{aligned} \gamma^0(R) &\stackrel{\text{def}}{=} \forall \mathbf{x} (\bigwedge_{i=l+1}^m A_i^0 \wedge \bigwedge_{i=m+1}^n \neg A_i^1 \rightarrow \bigvee_{i=1}^k A_i^0 \vee \bigvee_{i=k+1}^l \neg A_i^1), \\ \gamma^1(R) &\stackrel{\text{def}}{=} \forall \mathbf{x} (\bigwedge_{i=l+1}^m A_i^1 \wedge \bigwedge_{i=m+1}^n \neg A_i^1 \rightarrow \bigvee_{i=1}^k A_i^1 \vee \bigvee_{i=k+1}^l \neg A_i^1). \end{aligned}$$

For a logic program P define the first-order formula $\gamma(P)$ as

$$\gamma(P) \stackrel{\text{def}}{=} \bigwedge_{R \in P} \gamma^0(R) \wedge \bigwedge_{R \in P} \gamma^1(R).$$

and define the first-order formula S_P as

$$S_P \stackrel{\text{def}}{=} \bigwedge_{p \in \text{Pred}(P)} \forall \mathbf{x} (p^0(\mathbf{x}) \rightarrow p^1(\mathbf{x})),$$

where variables \mathbf{x} match the arity of p .

Using the transformation γ and the formula S_P we can express strong equivalence as an equivalence in first-order logic.

Proposition 2 ([38]). *Under the stable model semantics two logic programs P and Q are strongly equivalent iff the following equivalence holds in classical first-order logic $S_{P \cup Q} \wedge \gamma(P) \equiv S_{P \cup Q} \wedge \gamma(Q)$.*

2.3 Definition Synthesis with Craig Interpolation

A formula $Q(\mathbf{x})$ is *implicitly definable* in terms of a vocabulary (set of predicate and function symbols) V within a sentence F if, whenever two models of F agree on values of symbols in V , then they agree on the extension of Q , i.e., on the tuples of domain members that satisfy Q . In the special case where Q has no free variables, this means that they agree on the truth value of Q . Implicit definability can be expressed as

$$F \wedge F' \models \forall \mathbf{x} (Q(\mathbf{x}) \leftrightarrow Q'(\mathbf{x})), \quad (\text{i})$$

² With the notation γ we follow [25]. In [38, 39] σ^* is used for the same translation.

where F' and Q' are copies of F and Q with all symbols not in V replaced by fresh symbols. This semantic notion contrasts with the syntactic one of *explicit definability*: A formula $Q(\mathbf{x})$ is *explicitly definable* in terms of a vocabulary V within a sentence F if there exists a formula $R(\mathbf{x})$ in the vocabulary V such that

$$F \models \forall \mathbf{x} (R(\mathbf{x}) \leftrightarrow Q(\mathbf{x})). \quad (\text{ii})$$

The “Beth property” [7] states equivalence of both notions and applies to first-order logic. “Craig interpolation” is a tool that can be applied to prove the “Beth property” [21], and, moreover to construct formulas R from given F, Q, V from a proof of implicit definability. Craig’s interpolation theorem [20] applies to first-order logic and states that if a formula F entails a formula G (or, equivalently, that $F \rightarrow G$ is valid), then there exists a formula H , a *Craig interpolant* of F and G , with the properties that $F \models H$, $H \models G$, and the vocabulary of H (predicate and function symbols as well as free variables) is in the common vocabulary of F and G . Craig’s theorem can be strengthened to the existence of *Craig-Lyndon interpolants* [56] that satisfy the additional property that predicates in H occur only in polarities in which they occur in both F and G . In our technical framework this condition is expressed as $\text{Pred}^\pm(H) \subseteq \text{Pred}^\pm(F) \cap \text{Pred}^\pm(G)$.

Craig’s interpolation theorem can be proven by constructing H from a proof of $F \models G$. This works on the basis of sequent calculi [68,71] and analytic tableaux [29]. For calculi from automated first-order reasoning various approaches have been considered [10,40,44,67,76,77]. A method [76] for clausal tableaux [46] performs Craig-Lyndon interpolation and operates on proofs emitted by a general first-order prover, without need to modify the prover for interpolation, and inheriting completeness for full first-order logic from it. Indirectly that method also works on resolution proofs expanded into trees [77].

Observe that the characterization of implicit definability (i) can also be expressed as $F \wedge Q(\mathbf{x}) \models F' \rightarrow Q'(\mathbf{x})$. An “explicit” definiens $R(\mathbf{x})$ can now be constructed from given F , $Q(\mathbf{x})$ and V just as a Craig interpolant of $F \wedge Q(\mathbf{x})$ and $F' \rightarrow Q'(\mathbf{x})$. The synthesis of definitions by Craig interpolation was recognized as a logic-based core technique for view-based query reformulation in relational databases [5,59,66,72]. Often strengthened variations of Craig-Lyndon interpolation are used there that preserve criteria for domain independence, e.g., through relativized quantifiers [5] or range-restriction [77].

3 Variations of Craig Interpolation and Beth Definability for Logic Programs

We synthesize logic programs according to a variation of Beth’s theorem justified by a variation of Craig-Lyndon interpolation. Craig-Lyndon interpolation “out of the box” is not sufficient for our purposes: To obtain logic programs as definienda we need as a basis a stronger form of interpolation where the interpolant is not just a first-order formula but, moreover, is the first-order encoding of a logic program, permitting to actually extract the program.

3.1 Extracting Logic Programs from a First-Order Encoding

We address the questions of how to abstractly characterize first-order formulas that encode a logic program, and how to extract the program from a first-order formula that meets the characterization. As specified in Sect. 2.1, we assume first-order formulas over predicates superscripted with 0 and 1. The notation S_P (Definition 1) is now overloaded for 0/1-superscripted formulas F as $S_F \stackrel{\text{def}}{=} \bigwedge_{p \in \text{Pred}^{LP}(F)} \forall \mathbf{x} (p^0(\mathbf{x}) \rightarrow p^1(\mathbf{x}))$, where variables \mathbf{x} match the arity of p . We introduce a convenient notation for the result of systematically renaming all 0-superscripted predicates to their 1-superscripted correspondence.

Definition 3. For 0/1-superscripted first-order formulas F define $\text{rename}_{0 \rightarrow 1}(F)$ as F with all occurrences of 0-superscripted predicates p^0 replaced by the corresponding 1-superscripted predicate p^1 .

Obviously $\text{rename}_{0 \rightarrow 1}$ can be moved over logic operators, e.g., $\text{rename}_{0 \rightarrow 1}(F \wedge G) \equiv \text{rename}_{0 \rightarrow 1}(F) \wedge \text{rename}_{0 \rightarrow 1}(G)$, and $\text{rename}_{0 \rightarrow 1}(\forall \mathbf{x} F) \equiv \forall \mathbf{x} \text{rename}_{0 \rightarrow 1}(F)$. Semantically, $\text{rename}_{0 \rightarrow 1}$ preserves entailment and thus also equivalence.

Proposition 4. For 0/1-superscripted first-order formulas F and G it holds that (i) If $F \models G$, then $\text{rename}_{0 \rightarrow 1}(F) \models \text{rename}_{0 \rightarrow 1}(G)$. (ii) If $F \equiv G$, then $\text{rename}_{0 \rightarrow 1}(F) \equiv \text{rename}_{0 \rightarrow 1}(G)$.

Observe that for all rules R it holds that $\gamma^1(R) = \text{rename}_{0 \rightarrow 1}(\gamma^0(R))$ and for all logic programs P it holds that $\gamma(P) \models \text{rename}_{0 \rightarrow 1}(\gamma(P))$. On the basis of $\text{rename}_{0 \rightarrow 1}$ we define a first-order criterion for a formula encoding a logic program, that is, a first-order entailment that holds if and only if a given first-order formula encodes a logic program.

Definition 5. A 0/1-superscripted first-order formula F is said to *encode a logic program* iff F is universal and $S_F \wedge F \models \text{rename}_{0 \rightarrow 1}(F)$.

This criterion adequately characterizes that the formula represents a logic program on the basis of the translation γ . The following theorem justifies this.

Theorem 6 (Formulas Encoding a Logic Program). (i) For all logic programs P it holds that $\gamma(P)$ is a 0/1-superscripted first-order formula that encodes a logic program. (ii) If a 0/1-superscripted first-order formula F encodes a logic program, then there exists a logic program P such that $S_F \models \gamma(P) \leftrightarrow F$, $\text{Pred}(P) \subseteq \text{Pred}^{LP}(F)$ and $\text{Fun}(P) \subseteq \text{Fun}(F)$. Moreover, such a program P can be effectively constructed from F .

Proof. (6.i) Immediate from the definition of γ . (6.ii) Procedure 7 specified below and proven correct in Proposition 8 shows the construction of a suitable program. \square

Theorem 6.ii claims that the vocabulary of the program is only *included* in the respective vocabulary of the formula. This gives for the formula the freedom of a larger vocabulary with symbols that may be eliminated, e.g., by simplifications.

Procedure 7 (Decoding an Encoding of a Logic Program).

INPUT: A 0/1-superscripted first-order formula F that encodes a logic program.

METHOD:

1. Bring F into conjunctive normal form matching $\forall \mathbf{x} (M_0 \wedge M_1)$, where M_0 and M_1 are clausal formulas such that in all clauses of M_0 there is a literal whose predicate has superscript 0 and in all clauses of M_1 the predicates of all literals have superscript 1.
2. Partition M_1 into two clausal formulas M_1' and M_1'' such that

$$\forall \mathbf{x} \text{rename}_{0 \rightarrow 1}(M_0) \models \forall \mathbf{x} M_1''.$$

A possibility is to take $M_1' = M_1$ and $M_1'' = \top$. Another possibility that often leads to a smaller M_1' is to consider each clause C in M_1 and place it into M_1'' or M_1' depending on whether there is a clause D in M_0 such that $\text{rename}_{0 \rightarrow 1}(D)$ subsumes C .

3. Let P be the set of rules

$$A_1; \dots; A_k; \mathbf{not} A_{k+1}; \dots; \mathbf{not} A_l \leftarrow A_{l+1}, \dots, A_m, \mathbf{not} A_{m+1}, \dots, \mathbf{not} A_n$$

for each clause $\bigwedge_{i=l+1}^m A_i^0 \wedge \bigwedge_{i=m+1}^n \neg A_i^1 \rightarrow \bigvee_{i=0}^k A_i^0 \vee \bigvee_{i=k+1}^l \neg A_i^1$ in $M_0 \wedge M_1'$.

OUTPUT: Return P , a logic program such that $S_P \models \gamma(P) \leftrightarrow F$, $\text{Pred}(P) \subseteq \text{Pred}^{LP}(F)$ and $\text{Fun}(P) \subseteq \text{Fun}(F)$.

Proposition 8. *Procedure 7 is correct.*

Proof. For an input formula F and output program P the syntactic requirements $\text{Pred}(P) \subseteq \text{Pred}^{LP}(F)$ and $\text{Fun}(P) \subseteq \text{Fun}(F)$ are easy to see from the construction of P by the procedure. To prove the semantic requirement $S_F \models \gamma(P) \leftrightarrow F$ we first note the following assumptions, which follow straightforwardly from the specification of the procedure.

- (1) $S_F \wedge F \models \text{rename}_{0 \rightarrow 1}(F)$.
- (2) $F \equiv \forall \mathbf{x} (M_0 \wedge M_1' \wedge M_1'')$.
- (3) $\text{rename}_{0 \rightarrow 1}(\forall \mathbf{x} M_0) \models \forall \mathbf{x} M_1''$.
- (4) $\gamma(P) \equiv \forall \mathbf{x} (M_0 \wedge \text{rename}_{0 \rightarrow 1}(M_0) \wedge M_1')$.

The semantic requirement $S_F \models \gamma(P) \leftrightarrow F$ can then be derived as follows.

- | | |
|---|-----------------|
| (5) $S_F \wedge F \models \text{rename}_{0 \rightarrow 1}(\forall \mathbf{x} M_0)$. | By (2) and (1). |
| (6) $S_F \wedge F \models \forall \mathbf{x} (M_0 \wedge \text{rename}_{0 \rightarrow 1}(M_0) \wedge M_1')$. | By (5) and (2). |
| (7) $S_F \wedge F \models \gamma(P)$. | By (6) and (4). |
| (8) $\gamma(P) \models \forall \mathbf{x} (M_0 \wedge M_1'' \wedge M_1')$. | By (4) and (3). |
| (9) $\gamma(P) \models F$. | By (8) and (2). |
| (10) $S_F \models \gamma(P) \leftrightarrow F$. | By (9) and (7). |

□

Some examples illustrate Procedure 7 and the *encoding a logic program* property.

Example 9.

- (i) Consider the following clauses and programs.

$$\begin{array}{lll}
 C_1 = \neg p^0 \vee q^1 \vee r^0 & P = r \leftarrow p, \mathbf{not} \ q. & P' = r \leftarrow p, \mathbf{not} \ q. \\
 C_2 = \neg s^1 \vee t^1 \vee u^1 & \mathbf{not} \ s \leftarrow \mathbf{not} \ t, \mathbf{not} \ u. & \mathbf{not} \ s \leftarrow \mathbf{not} \ t, \mathbf{not} \ u. \\
 C_3 = \neg p^1 \vee q^1 \vee r^1 & \mathbf{not} \ p \leftarrow \mathbf{not} \ q, \mathbf{not} \ r. &
 \end{array}$$

Assume as input of Procedure 7 the formula $F = C_1 \wedge C_2 \wedge C_3$. Then $M_0 = C_1$ and $M_1 = C_2 \wedge C_3$. If in step 2 we set $M'_1 = M_1$ and $M''_1 = \top$, then the extracted program is P . We might, however, also set $M'_1 = C_2$ and $M''_1 = C_3$ and obtain the shorter strongly equivalent program P' .

- (ii) For $F = \neg p^1 \vee q^1 \vee r^0$ we have to set $M'_1 = M''_1 = M_1 = \top$ and the procedure yields $\{r; \mathbf{not} \ p \leftarrow \mathbf{not} \ q.\}$.
- (iii) The formula $F = \neg p^0 \vee q^1 \vee r^0$ does not encode a logic program because $S_F \wedge F \not\models \text{rename}_{0 \mapsto 1}(F)$.

Remark 1. For the extracted program it is desirable that it is not unnecessarily large. Specifically it should not contain rules that are easily identified as redundant. Step 2 of Procedure 7 permits techniques to keep M' small. Other possibilities include well-known formula simplification techniques that preserve equivalence such as removal of tautological or subsumed clauses and may be integrated into classification, step 1 of the procedure. In addition, conversions that just preserve equivalence modulo S_F may be applied, conceptually as a preprocessing step, although in practice possibly implemented on the clausal form. Procedure 7 then receives as input not F but a universal first-order formula F' whose vocabulary is included in that of F with the property

$$S_F \models F' \leftrightarrow F. \quad (\text{iii})$$

Formula F' then also encodes a program: That $S_{F'} \wedge F' \models \text{rename}_{0 \mapsto 1}(F')$ follows from $S_F \wedge F \models \text{rename}_{0 \mapsto 1}(F)$, (iii) and Proposition 4.ii. Procedure 7 guarantees for its output $S_{F'} \models \gamma(P) \leftrightarrow F'$, hence by (iii) it follows that $S_F \models \gamma(P) \leftrightarrow F$.

Example 10. Consider the following clauses and programs.

$$\begin{array}{lll}
 C_1 = \neg p^0 \vee q^1 \vee r^0 & P = r \leftarrow p, \mathbf{not} \ q. & P' = r \leftarrow p, \mathbf{not} \ q. \\
 C_2 = \neg p^0 \vee q^1 \vee r^1 & \leftarrow p, \mathbf{not} \ q, \mathbf{not} \ r. & \\
 C_3 = \neg p^1 \vee q^1 \vee r^1 & &
 \end{array}$$

Assume as input of Procedure 7 the formula $F = C_1 \wedge C_2 \wedge C_3$. Then $M_0 = C_1 \wedge C_2$ and $M_1 = C_3$, and, aiming at a short program, we can set $M'_1 = \top$ and $M''_1 = C_3$. The extracted program is then P . By preprocessing F according to Remark 1 we can eliminate C_2 from F and obtain the shorter strongly equivalent program P' .

3.2 A Refinement of Craig Interpolation for Logic Programs

With the material from Sect. 3.1 on extracting logic programs from formulas we can now state a theorem on *LP-interpolation*, where *LP* stands for *logic program*. It is a variation of Craig interpolation applying to first-order formulas that encode logic programs. The theorem states not only the existence of an LP-interpolant, but, moreover, also claims effective construction.

Theorem 11 (LP-Interpolation). *Let F be a 0/1-superscripted first-order formula that encodes a logic program and let G be a 0/1-superscripted first-order formula such that $\text{Fun}(F) \subseteq \text{Fun}(G)$ and $\mathbf{S}_F \wedge F \models \mathbf{S}_G \rightarrow G$. Then there exists a 0/1-superscripted first-order formula H , called the LP-interpolant of F and G , such that*

1. $\mathbf{S}_F \wedge F \models H$.
2. $H \models \mathbf{S}_G \rightarrow G$.
3. $\text{Pred}^\pm(H) \subseteq S \cup \{+p^1 \mid +p^0 \in S\} \cup \{-p^1 \mid -p^0 \in S\}$, where $S = \text{Pred}^\pm(\mathbf{S}_F \wedge F) \cap \text{Pred}^\pm(\mathbf{S}_G \rightarrow G)$.
4. $\text{Fun}(H) \subseteq \text{Fun}(F)$.
5. H encodes a logic program.

Moreover, if existence holds, then an LP-interpolant H can be effectively constructed, via a universal Craig-Lyndon interpolant of $F \wedge \mathbf{S}_F$ and $\mathbf{S}_G \rightarrow G$.

Proof. We show the construction of a suitable formula H . Let H' be a Craig-Lyndon interpolant of $\mathbf{S}_F \wedge F$ and $\mathbf{S}_G \rightarrow G$. Since F and \mathbf{S}_F are universal first-order formulas and we have the precondition $\text{Fun}(F) \subseteq \text{Fun}(G)$, we may in addition assume that H' is a universal first-order formula. (This additional condition is guaranteed for example by the interpolation method from [76], which computes H' directly from a clausal tableaux proof, or indirectly from a resolution proof [77].) Define $H \stackrel{\text{def}}{=} H' \wedge \text{rename}_{0 \rightarrow 1}(H')$. Claims 2, 4 and 5 of the theorem statement are then easy to see. Claim 1 can be shown as follows. We may assume the following.

- | | |
|--|---|
| (1) $\mathbf{S}_F \wedge F \models H'$. | Since H' is a Craig-Lyndon interpolant. |
| (2) $\mathbf{S}_F \wedge F \models \text{rename}_{0 \rightarrow 1}(F)$. | Since F encodes a logic program. |

Claim 1 can then be derived in the following steps.

- | | |
|--|---|
| (3) $\text{rename}_{0 \rightarrow 1}(\mathbf{S}_F \wedge F) \models \text{rename}_{0 \rightarrow 1}(H')$. | By (1) and Proposition 4.i. |
| (4) $\mathbf{S}_F \wedge F \models \text{rename}_{0 \rightarrow 1}(\mathbf{S}_F \wedge F)$. | By (2), since $\text{rename}_{0 \rightarrow 1}(\mathbf{S}_F) \equiv \top$. |
| (5) $\mathbf{S}_F \wedge F \models \text{rename}_{0 \rightarrow 1}(H')$. | By (4) and (3). |
| (6) $\mathbf{S}_F \wedge F \models H$. | By (5) and (1), since $H = H' \wedge \text{rename}_{0 \rightarrow 1}(H')$. |

Claim 3 follows because since H' is a Craig-Lyndon interpolant it holds that $\text{Pred}^\pm(H') \subseteq \text{Pred}^\pm(\mathbf{S}_F \wedge F) \cap \text{Pred}^\pm(\mathbf{S}_G \rightarrow G)$. With the predicate occurrences in $\text{rename}_{0 \rightarrow 1}(H')$, i.e., 1-superscripted predicates in positions of 0-superscripted predicates in H' , we obtain the restriction of $\text{Pred}^\pm(H)$ stated as claim 3. \square

For an LP-interpolant of formulas F and G , where F encodes a logic program, the semantic properties stated in claims 1 and 2 are those of a Craig or Craig-Lyndon interpolant of $\mathbf{S}_F \wedge F$ and $\mathbf{S}_G \rightarrow G$. The allowed polarity/predicate pairs are those common in $\mathbf{S}_F \wedge F$ and $\mathbf{S}_G \rightarrow G$, as in a Craig-Lyndon interpolant, and, in addition, the 1-superscripted versions of polarity/predicate pairs that appear only 0-superscripted among these common pairs. These additional pairs are those that might occur in the result of $\text{rename}_{0 \rightarrow 1}$ applied to a Craig-Lyndon interpolant. In contrast to a Craig interpolant, functions are only constrained

by the first given formula F . Permitting only functions common to F and G can result in an interpolant with existential quantification, which thus does not encode a program. Claim 5 states that the LP-interpolant indeed encodes a logic program as characterized in Definition 5. This property is, so-to-speak, passed through from the given formula F to the LP-interpolant.

3.3 Effective Projective Definability of Logic Programs

We present a variation of the “Beth property” that applies to logic programs with stable model semantics and takes strong equivalence into account. The underlying technique is our LP-interpolation, Theorem 11. It maps into Craig-Lyndon interpolation for first-order logic, utilizing that strong equivalence of logic programs can be expressed as first-order equivalence of encoded programs. This approach allows the effective construction of logic programs in the role of “explicit definitions” via Craig-Lyndon interpolation on first-order formulas. Our variation of the “Beth property” is *projective* as it is with respect to a given set of predicates allowed in the definiens. While our LP-interpolation theorem was expressed in terms of first-order formulas that encode logic programs, we now phrase definability entirely in terms of logic programs.³

Theorem 12 (Effective Projective Definability of Logic Programs). *Let P and Q be logic programs and let $V \subseteq \text{Pred}(P) \cup \text{Pred}(Q)$ be a set of predicates. The existence of a logic program R with $\text{Pred}(R) \subseteq V$, $\text{Fun}(R) \subseteq \text{Fun}(P) \cup \text{Fun}(Q)$ such that $P \cup R$ and $P \cup Q$ are strongly equivalent is expressible as entailment between two first-order formulas. Moreover, if existence holds, such a program R can be effectively constructed, via a universal Craig-Lyndon interpolant of the left and the right side of the entailment.*

Proof. The first-order entailment that characterizes the existence of a logic program R is $\mathbf{S}_P \wedge \mathbf{S}_Q \wedge \gamma(P) \wedge \gamma(Q) \models \neg \mathbf{S}_{P'} \vee \neg \mathbf{S}_{Q'} \vee \neg \gamma(P') \vee \gamma(Q')$, where the primed programs P' and Q' are like P and Q , except that predicates not in V are replaced by fresh predicates. If the entailment holds, we can construct a program R as follows: Let H be the LP-interpolant of $\gamma(P) \wedge \gamma(Q)$ and $\neg \gamma(P') \vee \gamma(Q')$, as specified in Theorem 11, and extract the program R from H with Procedure 7. That R constructed in this way has the properties claimed in the theorem statement can be shown as follows. Since H is an LP-interpolant it follows that

- (1) $\mathbf{S}_P \wedge \mathbf{S}_Q \wedge \gamma(P) \wedge \gamma(Q) \models H$.
- (2) $H \models \neg \mathbf{S}_{P'} \vee \neg \mathbf{S}_{Q'} \vee \neg \gamma(P') \vee \gamma(Q')$.
- (3) $\text{Pred}^{LP}(H) \subseteq V$.
- (4) $\text{Fun}(H) \subseteq \text{Fun}(P) \cup \text{Fun}(Q)$.
- (5) H encodes a logic program.

From the preconditions of the theorem and since R is extracted from H with Procedure 7 and thus meets the properties stated in Theorem 6.ii it follows that

³ LP-interpolation could also be phrased in terms of logic programs, providing an interpolation result for logic programs on its own, not just as basis for definability. We plan to address this in future work.

- (6) $V \subseteq \text{Pred}(P) \cup \text{Pred}(Q)$.
- (7) $S_H \models \gamma(R) \leftrightarrow H$.
- (8) $\text{Pred}(R) \subseteq \text{Pred}^{LP}(H)$.
- (9) $\text{Fun}(R) \subseteq \text{Fun}(H)$.

The claimed properties of the theorem statement can then be derived as steps (10), (11) and (18) as follows.

- (10) $\text{Pred}(R) \subseteq V$. By (8) and (3).
- (11) $\text{Fun}(R) \subseteq \text{Fun}(P) \cup \text{Fun}(Q)$. By (9) and (4).
- (12) $S_P \wedge S_Q \models S_H$. By (6) and (3).
- (13) $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q) \models \gamma(R)$. By (12), (7) and (1).
- (14) $S_{P'} \wedge S_{Q'} \wedge \gamma(P') \wedge \neg\gamma(Q') \models \neg H$. By (2).
- (15) $S_P \wedge S_Q \wedge \gamma(P) \wedge \neg\gamma(Q) \models \neg H$. By (14), (3) and (4).
- (16) $S_P \wedge S_Q \wedge \gamma(P) \wedge \neg\gamma(Q) \models \neg\gamma(R)$. By (15), (12) and (7).
- (17) $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(R) \equiv S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q)$. By (16) and (13).
- (18) $P \cup R$ and $P \cup Q$ are strongly equivalent. By (17) and Proposition 2.

Conversely, we have to show that if there exists a logic program R with the properties in the above theorem statement, then the characterizing entailment $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q) \models \neg S_{P'} \vee \neg S_{Q'} \vee \neg\gamma(P') \vee \gamma(Q')$ does hold. We may assume

- (19) $\text{Pred}(R) \subseteq V \subseteq \text{Pred}(P) \cup \text{Pred}(Q)$.
- (20) $\text{Fun}(R) \subseteq \text{Fun}(P) \cup \text{Fun}(Q)$.
- (21) $P \cup R$ and $P \cup Q$ are strongly equivalent.

The characterizing entailment can then be derived as follows.

- (22) $S_P \wedge S_Q \models S_R$ By (19).
- (23) $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(R) \equiv S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q)$. By (21), Proposition 2 and (22).
- (24) $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q) \models \gamma(R)$. By (23).
- (25) $S_P \wedge S_Q \wedge \gamma(P) \wedge \neg\gamma(Q) \models \neg\gamma(R)$. By (23).
- (26) $S_{P'} \wedge S_{Q'} \wedge \gamma(P') \wedge \neg\gamma(Q') \models \neg\gamma(R)$. By (25), (19) and (20).
- (27) $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q) \models \neg S_{P'} \vee \neg S_{Q'} \vee \neg\gamma(P') \vee \gamma(Q')$. By (26) and (24).

□

We now give some examples from the application point of view.

Example 13. The following examples show for given programs P, Q and sets V of predicates a possible value of R according to Theorem 12.

- (i) $Q = \mathbf{p} \leftarrow \mathbf{q}, \mathbf{r}$. $V = \{\mathbf{p}, \mathbf{r}\}$ $R = \mathbf{p} \leftarrow \mathbf{r}$.
 $\mathbf{p}; \mathbf{q} \leftarrow \mathbf{r}$.
 $\mathbf{q} \leftarrow \mathbf{q}, \mathbf{s}$.

In this first example we consider the special case where P is empty and thus not shown. Predicates \mathbf{q} and \mathbf{s} are redundant in Q , “absolutely” and not just relative to a program P . By Theorem 12, this is proven with the characterizing first-order entailment and, moreover, a strongly equivalent reformulation of Q without \mathbf{q} and \mathbf{s} is obtained as R .

- (ii) $P = p(X) \leftarrow q(X)$. $Q = r(X) \leftarrow p(X)$. $V = \{p, r\}$ $R = r(X) \leftarrow p(X)$.
 $r(X) \leftarrow q(X)$.

Only r and p are allowed in R . Or, equivalently, q is redundant in Q , *relative* to program P . Again, this is proven with the characterizing first-order entailment and, moreover, a strongly equivalent reformulation of Q without q is obtained as R . It is the clause in Q with q that is redundant relative to P and hence is eliminated in R .

- (iii) $P = \leftarrow p(X), q(X)$. $Q = r(X) \leftarrow p(X), \text{not } q(X)$. $V = \{p, r\}$
 $R = r(X) \leftarrow p(X)$.

Only r and p are allowed in R . The negated literal with q in the body of the rule in Q is redundant relative to P and is eliminated in R .

- (iv) $P = p(X) \leftarrow q(X), \text{not } r(X)$. $Q = t(X) \leftarrow p(X)$. $V = \{q, r, s, t\}$
 $p(X) \leftarrow s(X)$. $R = t(X) \leftarrow q(X), \text{not } r(X)$.
 $\text{not } r(X); s(X) \leftarrow p(X)$. $t(X) \leftarrow s(X)$.
 $q(X); s(X) \leftarrow p(X)$.

The predicate p is not allowed in R . The idea is that p is a predicate that can be used by a client but is not in the actual knowledge base. Program P expresses a schema mapping from the client predicate p to the knowledge base predicates q, r, s . The result program R is a rewriting of the client query Q in terms of knowledge base predicates. Only the first two rules of P actually describe the mapping. The other two rules complete them to a full definition, similar to Clark's completion [19], but here yielding also a program. Such completed predicate specifications seem necessary for the envisaged reformulation tasks.

- (v) $P = \text{As in Example 13.iv}$. $Q = t(X) \leftarrow q(X), \text{not } r(X)$. $V = \{p, t\}$
 $t(X) \leftarrow s(X)$.

$$R = t(X) \leftarrow p(X).$$

In this example P is from Example 13.iv, while Q and R are also from that example but switched. The vocabulary allows only p and t . While Example 13.iv realizes an unfolding of p , this example realizes folding into p .

- (vi) $P = n(X) \leftarrow z(X)$. $Q = \text{not } n(X_2) \leftarrow z(X_0), s(X_0, X_1)$,
 $n(X) \leftarrow n(Y), s(Y, X)$. $s(X_1, X_2)$.

$$V = \{z, s\} \quad R = \leftarrow z(X_0), s(X_0, X_1), s(X_1, X_2).$$

Program P defines natural numbers recursively. Program Q has a rule whose body specifies the natural number 2 and whose head denies that 2 is a natural number. Because P implies that 2 is a natural number, this head is in R rewritten to the empty head, enforced by disallowing the predicate for natural numbers in R .

- (vii) $P = c(X, Y, Z) \leftarrow r(X, Y), r(Y, Z)$. $Q = r(X, Y); \text{not } r(X, Y)$.
 $\leftarrow c(X, Y, Z), \text{not } r(X, Y)$. $\leftarrow c(X, Y, Z), \text{not } r(X, Z)$.
 $\leftarrow c(X, Y, Z), \text{not } r(Y, Z)$. $R = r(X, Y); \text{not } r(X, Y)$.

$$V = \{r\} \quad \leftarrow r(X, Y), r(Y, Z), \text{not } r(X, Z).$$

Program Q describes a transitive relation r using the helper predicate c to identify chains where transitivity needs to be checked. In R the use of c is

not allowed, program P gives the definition of c . Similar to Example 13.iv this realizes an unfolding of c .

Definability according to Theorem 12 inherits potential *decidability* from the first-order entailment problem that characterizes it. If, e.g., in the involved programs only constants occur as function symbols, the characterizing entailment can be expressed as validity in the decidable Bernays-Schönfinkel class.

3.4 Constraining Positions of Predicates Within Rules

The sensitivity of LP-interpolation to polarity inherited from Craig-Lyndon interpolation and the program encoding with superscripted predicates offers a more fine-grained control of the vocabulary of definitions than Theorem 12 by considering also positions of predicates in rules. The following corollary shows this.

Corollary 14 (Position-Constrained Effective Projective Definability of Logic Programs). *Let P and Q be logic programs and let $V_+, V_{+1}, V_- \subseteq \text{Pred}(P) \cup \text{Pred}(Q)$ be three sets of predicates. Call a logic program R in scope of $\langle V_+, V_{+1}, V_- \rangle$ if predicates p occur in R only as specified in the following table.*

p is allowed in	only if p is in
Positive heads	V_+
Negative bodies	$V_+ \cup V_{+1}$
Negative heads	V_-
Positive bodies	V_-

The existence of a logic program R in scope of $\langle V_+, V_{+1}, V_- \rangle$ with $\text{Fun}(R) \subseteq \text{Fun}(P) \cup \text{Fun}(Q)$ such that $P \cup R$ and $P \cup Q$ are strongly equivalent is expressible as entailment between two first-order formulas. Moreover, if existence holds, such a program R can be effectively constructed, via a universal Craig-Lyndon interpolant of the left and the right side of the entailment.

Proof (Sketch). Like Theorem 12 but with applying the renaming of disallowed predicates not already at the program level but in the first-order encoding with considering polarity. Let V^\pm be the set of polarity/predicate pairs defined as $V^\pm \stackrel{\text{def}}{=} \{+p^0 \mid p \in V_+\} \cup \{+p^1 \mid p \in V_+ \cup V_{+1}\} \cup \{-p^0 \mid p \in V_-\} \cup \{-p^1 \mid p \in V_-\}$. The corresponding entailment underlying definability and LP-interpolation is then $S_P \wedge S_Q \wedge \gamma(P) \wedge \gamma(Q) \models \neg S'_P \vee \neg S'_Q \vee \neg \gamma(P)' \vee \gamma(Q)' \vee \neg \text{Aux}$, where the primed variations of formulas are obtained by replacing each predicate p that does not appear in V^\pm or appears in V^\pm with only a single polarity by a dedicated fresh predicate p' . (Note that negation and priming commute.) With $W \stackrel{\text{def}}{=} \text{Pred}^\pm(\neg S_P \vee \neg S_Q \vee \neg \gamma(P) \vee \gamma(Q))$ we define Aux as

$$+p \in V^\pm, -p \notin V^\pm, +p \in W \quad \bigwedge \quad \forall \mathbf{x} (p(\mathbf{x}) \rightarrow p'(\mathbf{x})) \quad \wedge \quad \bigwedge \quad \forall \mathbf{x} (p'(\mathbf{x}) \rightarrow p(\mathbf{x})),$$

$$-p \in V^\pm, +p \notin V^\pm, -p \in W$$

where \mathbf{x} matches the arity of the respective predicates p . □

In general, for first-order formulas F, G the second-order entailment $F \models \forall p G$, where p is a predicate, holds if and only if the first-order entailment $F \models G'$ holds, where G' is G with p replaced by a fresh predicate p' . This explains the construction of the right side of the entailment in the proof of Corollary 14 as an encoding of quantification upon (or “forgetting about”) a predicate only in a *single polarity*. With predicate quantification this can be expressed, e.g., for positive polarity, as $\exists +p G \stackrel{\text{def}}{=} \exists p' (G' \wedge \forall \mathbf{x} (p(\mathbf{x}) \rightarrow p'(\mathbf{x})))$, where p' is a fresh predicate and G' is G with p replaced by p' . We illustrate the specification of Aux with an example.

Example 15. Assume that $+p \in V^\pm$ and $-p \notin V^\pm$. Let G stand for $S_P \wedge S_Q \wedge \gamma(P) \wedge \neg\gamma(Q)$ and let G' stand for G with all occurrences of p replaced by p' . For now we ignore the restrictions of Aux by W as they only have a heuristic purpose. The following statements, which are all equivalent to each other, illustrate the specification of Aux in a step-by-step fashion. We start with expressing the required “forgetting”. Since it appears in a negation, we have to forget here the “allowed” $+p$. (1) $F \models \neg\exists +p G$. (2) $F \models \neg\exists p' (G' \wedge \forall \mathbf{x} (p(\mathbf{x}) \rightarrow p'(\mathbf{x})))$. (3) $F \models \forall p' (\neg G' \vee \neg\forall \mathbf{x} (p(\mathbf{x}) \rightarrow p'(\mathbf{x})))$. (4) $F \models \neg G' \vee \neg\forall \mathbf{x} (p(\mathbf{x}) \rightarrow p'(\mathbf{x}))$. (5) $F \models \neg G' \vee \neg Aux$.

Observing the restrictions by membership in W in the definition of Aux can result in a smaller formula Aux . Continuing the example, this can be illustrated as follows. Assume $+p \in V^\pm$ and $-p \notin V^\pm$ as before and in addition $+p \notin W$. Since $+p \notin W$ it follows that $-p \notin \text{Pred}^\pm(G)$. So “forgetting” about $+p$ in G is then just $\exists p' G'$ and the Aux component for p is not needed. (Also a further simplification, outlined in the remark below, is possible in this case.)

Remark 2. The view of “priming” as predicate quantification justifies a heuristically useful simplification of interpolation inputs for definability: If a predicate p occurs in a formula F only with positive (negative) polarity, then $\exists p F$ is equivalent to F with all atoms of the form $p(\mathbf{t})$ replaced by \top (\perp). Hence, if a predicate to be primed occurs only in a single polarity, we can replace all atoms with it by a truth value constant.

We now turn to application possibilities of Corollary 14. While it gives some control over the position of predicates, it does not allow to discriminate between allowing a predicate in negative heads and positive bodies. Predicates allowed in positive heads are also allowed in negative bodies. We give some examples.

Example 16. The following examples show for given programs P, Q and sets V_+, V_{+1}, V_- of predicates a possible value of R according to Corollary 14. In all the examples it is essential that a predicate is disallowed in R only in a single polarity. If it would not be allowed at all there would be no solution R .

$$(i) \quad \begin{array}{lll} P = p \leftarrow \mathbf{q}. & Q = r \leftarrow \mathbf{p}. & V_+ = \{p, q, r, s\} \\ & r \leftarrow \mathbf{q}. & V_{+1} = \{\} \\ & q \leftarrow \mathbf{s}. & V_- = \{p, r, s\} \end{array} \quad \begin{array}{l} R = r \leftarrow \mathbf{p}. \\ \mathbf{q} \leftarrow \mathbf{s}. \end{array}$$

Here \mathbf{q} is allowed in R in positive heads (and negative bodies) but not in positive bodies (and negative heads). Parentheses indicate constraints that apply but are not relevant for the example.

$$(ii) \quad P = p \leftarrow q. \quad Q = \leftarrow q, \mathbf{not} p. \quad V_+ = \{q, r, s\} \quad R = r \leftarrow q. \\ r \leftarrow q. \quad V_{+1} = \{\} \quad s \leftarrow p. \\ s \leftarrow p. \quad V_- = \{p, q, r, s\}$$

Here p is allowed in R in positive bodies (and negative heads) but not in negative bodies (and positive heads).

$$(iii) \quad P = p \leftarrow q. \quad Q = s \leftarrow \mathbf{not} r. \quad V_+ = \{s\} \quad R = s \leftarrow \mathbf{not} r. \\ r \leftarrow p. \quad r \leftarrow q. \quad V_{+1} = \{r\} \\ V_- = \{p, q, r, s\}$$

Here r is allowed in R in negative bodies and but not in positive heads.

4 Prototypical Implementation

We implemented the synthesis according to Theorem 12 and Corollary 14 prototypically with the PIE (*Proving, Interpolating, Eliminating*) environment [74, 75], which is embedded in SWI-Prolog [78]. The implementation and all requirements are free software, see <http://cs.christophwernhard.com/pie/asp>.

For Craig-Lyndon interpolation there are several options, yielding different solutions for some problems. Proving can be performed by *CMProver*, a clausal tableaux/connection prover connection [8, 9, 46] included in PIE, similar to PTPP [69], *SETHEO* [47] and *leanCoP* [61], or by *Prover9* [58]. Interpolant extraction is performed on clausal tableaux following [76]. Resolution proofs by *Prover9* are first translated to tableaux with the *hyper* property, which allows to pass range-restriction and the Horn property from inputs to outputs of interpolation [77]. Optionally also proofs by *CMProver* can be transformed before interpolant extraction to ensure the hyper property. With *CMProver* it is possible to enumerate alternate interpolants extracted from alternate proofs. More powerful provers such as *E* [65] and *Vampire* [43] unfortunately do not emit gap-free proofs that would be suited for extracting interpolants.

The organization of the implementation closely follows the abstract exposition in Sect. 3, with Prolog predicates corresponding to theorems. For convenience in some applications, the predicate that realizes Theorem 12 and Corollary 14 permits to specify the vocabulary also complementary, by listing predicates not allowed in the result. In general, if outputs are largely *semantically* characterized, *simplifications* play a key role. Solutions with redundancies should be avoided, even if they are correct. This concerns all stages of our workflow: preparation of the interpolation inputs, choice or transformation of the proof used for interpolant extraction, interpolant extraction, the interpolant itself, and the first-order representation of the output program, where strong equivalence must be preserved, possibly modulo a background program. Although our system offers various simplifications at these stages, this seems an area for improvement with large impact for practice. Some particular issues only show up with experiments. For example, for both *CMProver* and *Prover9* a preprocessing of the right sides of the interpolation entailments to reduce the number of distinct variables that are Skolemized by the systems was necessary, even for relatively small inputs.

The application of first-order provers to interpolation for reformulation tasks is a rather unknown territory. Experiments with limited success are described

in [4]. Our prototypical implementation covers the full range from the application task, synthesis of an answer set program for two given programs and a given vocabulary, to the actual construction of a result program via Craig-Lyndon interpolation by a first-order prover. At least for small inputs such as the examples in the paper it successfully produces results. We expect that with larger inputs from applications it at least helps to identify and narrow down the actual issues that arise for practical interpolation with current first-order proving technology. This is facilitated by the embedding into PIE, which allows easy interfacing to community standards, e.g., by exporting proving problems underlying interpolation in the *TPTP* [70] format.

5 Conclusion

We presented an effective variation of projective Beth definability based on Craig interpolation for answer set programs with respect to strong equivalence under the stable model semantics. Interpolation theorems for logic programs under stable models semantics were shown before in [1], where, however, programs are only on the left side of the entailment underlying interpolation, and the right side as well as the interpolant are just first-order formulas. Craig interpolation and Beth definability for answer set programs was considered in [31, 64], but with just existence results for equilibrium logic, which transfer to answer set semantics. The transfer of Craig interpolation and Beth definability from monotonic logics to default logics is investigated in [15], however, applicability to the stable model semantics and a relationship to strong equivalence are not discussed.

In [73] ontology-mediated querying over a knowledge base for specific description logics is considered, based on Beth definability and Craig interpolation. Interpolation is applied there to the Clark's completion [19] of a Datalog program. Although completion semantics is a precursor of the stable model semantics, both agreeing on a subclass of programs, completion seems applied in [73] actually on program fragments, or on the “schema level”, as in our Examples 13.iv, 13.v and 13.vii. A systematic investigation of these forms of completion is on our agenda. *Forgetting* [22, 37] or, closely related, *uniform interpolation* and *second-order quantifier elimination* [30], may be seen as generalizing Craig interpolation: an expression is sought that captures exactly what a given expression says about a restricted vocabulary. A *Craig* interpolant is moreover related to a second given expression entailed by the first, allowing to extract it from a proof of the entailment.

We plan to extend our approach to classes of programs with practically important language extensions. Arithmetic expressions and comparisons in rule bodies are permitted in the language mini-GRINGO, used already in recent works on verifying strong equivalence [24, 26, 51]. We considered strong equivalence relative to a context *program* P . These contexts might be generalized to first-order theories that capture theory extensions of logic programs [13, 33, 41].

So far, our approach characterizes result programs syntactically by restricting allowed predicates and, to some degree, also their positions in rules. Can this be generalized? Restricting allowed functions, including constants, seems not

possible: If a function occurs only in the left side of the entailment underlying Craig interpolation, the interpolant may have existentially quantified variables, making the conversion to a logic program impossible. From the interpolation side it is known that the Horn property and variations of range-restriction can be preserved [77]. It remains to be investigated, how this transfers to synthesizing logic programs, where in particular restrictions of the rule form and *safety* [14, 45], an important property related to range restriction, are of interest.

In addition to verifying strong equivalence, recent work addresses verifying further properties, e.g., *uniform* equivalence (equivalence under inputs expressed as ground facts) [24, 26, 52]. The approach is to use completion [19, 50] to express the verification problem in classical first-order logic. It is restricted to so-called *locally tight* logic programs [27]. Also forms of equivalence that abstract from “hidden” predicates are mostly considered for such restricted program classes, as relative equivalence [55], projected answer sets [23], or external behavior [24]. It remains future work to consider definability with uniform equivalence and hidden predicates, possibly using completion for translating logic programs to formulas (instead of γ), although it applies only to restricted classes of programs.

Independently from the application to program synthesis, our characterization of *encodes a program* and our procedure to extract a program from a formula suggest a novel practical method for transforming logic programs while preserving strong equivalence. The idea is as follows, where P is the given program: *First-order transformations* are applied to $F \stackrel{\text{def}}{=} \gamma(P)$ to obtain a first-order formula F' such that $S_F \wedge F' \equiv S_F \wedge F$. For transformations that result in a universal formula, F' encodes a logic program, as argued in Remark 1. Applying the extraction procedure to F' then results in a program P' that is strongly equivalent to P . This makes the wide range of known first-order simplifications and formula transformations applicable and provides a firm foundation for soundness of special transformations. We expect that this approach supplements known dedicated simplifications that preserve strong equivalence [11, 23].

With its background in artificial intelligence research, answer set programming is a declarative approach to problem solving, where specifications are processed by automated systems. It is suitable for meta-level reasoning to verify properties of specifications and to synthesize new specifications. On the basis of a technique to verify an equivalence property of answer set programs we developed a synthesis technique. Our tools were Craig interpolation and Beth definability, fundamental insights about first-order logic that relate given formulas to further formulas characterized in certain ways. Practically realized with automated first-order provers, Craig interpolation and Beth definability become tools to synthesize formulas, and, as shown here, also answer set programs.

Acknowledgments. The authors thank anonymous reviewers for helpful suggestions to improve the presentation.

References

1. Amir, E.: Interpolation theorems for nonmonotonic reasoning systems. In: Flesca, S., Greco, S., Ianni, G., Leone, N. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 233–244. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45757-7_20
2. Artale, A., Jung, J.C., Mazzullo, A., Ozaki, A., Wolter, F.: Living without Beth and Craig: definitions and interpolants in description and modal logics with nominals and role inclusions. *ACM Trans. Comp. Log.* **24**(4), 34:1–34:51 (2023). <https://doi.org/10.1145/3597301>
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2010)
4. Benedikt, M., Kostylev, E.V., Mogavero, F., Tsamoura, E.: Reformulating queries: theory and practice. In: Sierra, C. (ed.) IJCAI 2017, pp. 837–843. *ijcai.org* (2017). <https://doi.org/10.24963/ijcai.2017/116>
5. Benedikt, M., Leblay, J., ten Cate, B., Tsamoura, E.: Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation. Morgan & Claypool (2016). <https://doi.org/10.1007/978-3-031-01856-5>
6. Benedikt, M., Pradic, C., Wernhard, C.: Synthesizing nested relational queries from implicit specifications. In: PODS 2023, pp. 33–45. ACM (2023). <https://doi.org/10.1145/3584372.3588653>
7. Beth, E.W.: On Padoa’s method in the theory of definition. *Indag. Math.* **15**, 330–339 (1953)
8. Bibel, W.: Automated Theorem Proving. Vieweg, Braunschweig (1987). <https://doi.org/10.1007/978-3-322-90102-6>. First edition 1982
9. Bibel, W., Otten, J.: From Schütte’s formal systems to modern automated deduction. In: The Legacy of Kurt Schütte, pp. 217–251. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49424-7_13
10. Bonacina, M.P., Johansson, M.: On interpolation in automated theorem proving. *J. Autom. Reasoning* **54**(1), 69–97 (2015). <https://doi.org/10.1007/s10817-014-9314-0>
11. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Prog.* **40**(1), 1–46 (1999). [https://doi.org/10.1016/S0743-1066\(98\)10030-4](https://doi.org/10.1016/S0743-1066(98)10030-4)
12. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. *Theory Pract. Log. Program.* **7**(6), 745–759 (2007). <https://doi.org/10.1017/S1471068407003110>
13. Cabalar, P., Kaminski, R., Morkisch, P., Schaub, T.: *telingo* = ASP + time. In: Balduccini, M., Lierler, Y., Woltran, S. (eds.) LPNMR 2019. LNCS, vol. 11481, pp. 256–269. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20528-7_19
14. Cabalar, P., Pearce, D., Valverde, A.: A revised concept of safety for general answer set programs. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS (LNAI), vol. 5753, pp. 58–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04238-6_8
15. Cassano, V., Fervari, R., Areces, C., Castro, P.F.: Interpolation and Beth definability in default logics. In: Calimeri, F., Leone, N., Manna, M. (eds.) JELIA 2019. LNCS (LNAI), vol. 11468, pp. 675–691. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19570-0_44
16. ten Cate, B., Conradie, W., Marx, M., Venema, Y.: Definitorially complete description logics. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) KR 2006, pp. 79–89. AAAI Press (2006). <http://www.aaai.org/Library/KR/2006/kr06-011.php>

17. ten Cate, B., Franconi, E., Seylan, I.: Beth definability in expressive description logics. *JAIR* **48**, 347–414 (2013). <https://doi.org/10.1613/JAIR.4057>
18. Chen, Y., Lin, F., Li, L.: SELP – a system for studying strong equivalence between logic programs. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) *LPNMR 2005*. LNCS (LNAI), vol. 3662, pp. 442–446. Springer, Heidelberg (2005). https://doi.org/10.1007/11546207_43
19. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) *Logic and Data Bases*, vol. 1, pp. 293–322. Plenum Press, New York (1978)
20. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.* **22**(3), 250–268 (1957). <https://doi.org/10.2307/2963593>
21. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.* **22**(3), 269–285 (1957). <https://doi.org/10.2307/2963594>
22. Delgrande, J.P.: A knowledge level account of forgetting. *JAIR* **60**, 1165–1213 (2017). <https://doi.org/10.1613/JAIR.5530>
23. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer-set programming. In: Kaelbling, L.P., Saffiotti, A. (eds.) *IJCAI 2005*, pp. 97–102. Professional Book Center (2005). <http://ijcai.org/Proceedings/05/Papers/1177.pdf>
24. Fandinno, J., Hansen, Z., Lierler, Y., Lifschitz, V., Temple, N.: External behavior of a logic program and verification of refactoring. *Theory Pract. Log. Program.* **23**(4), 933–947 (2023). <https://doi.org/10.1017/S1471068423000200>
25. Fandinno, J., Lifschitz, V.: On Heuer’s procedure for verifying strong equivalence. In: Gaggl, S.A., Martinez, M.V., Ortiz, M. (eds.) *JELIA 2023*. LNCS, vol. 14281, pp. 253–261. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-43619-2_18
26. Fandinno, J., Lifschitz, V., Lühne, P., Schaub, T.: Verifying tight logic programs with anthem and Vampire. *Theory Pract. Log. Program.* **20**(5), 735–750 (2020). <https://doi.org/10.1017/S1471068420000344>
27. Fandinno, J., Lifschitz, V., Temple, N.: Locally tight programs. *Theory Pract. Log. Program.*, 1–31 (2024). <https://doi.org/10.1017/S147106842300039X>
28. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. *Artif. Intell.* **175**(1), 236–263 (2011). <https://doi.org/10.1016/J.ARTINT.2010.04.011>
29. Fitting, M.: *First-Order Logic and Automated Theorem Proving*, 2nd edn. Springer, New York (1995). <https://doi.org/10.1007/978-1-4612-2360-3>
30. Gabbay, D.M., Schmidt, R.A., Szałas, A.: *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, London (2008)
31. Gabbay, D.M., Pearce, D., Valverde, A.: Interpolable formulas in equilibrium logic and answer set programming. *JAIR* **42**, 917–943 (2011). <https://jair.org/index.php/jair/article/view/10743>
32. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. *Theory Pract. Log. Program* **15**(4–5), 449–463 (2015). <https://doi.org/10.1017/S1471068415000150>
33. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with Clingo 5. In: Carro, M., King, A., Saeedloei, N., Vos, M.D. (eds.) *ICLP 2016*. OASICS, vol. 52, pp. 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). <https://doi.org/10.4230/OASICS.ICLP.2016.2>
34. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with Clingo. *Theory Pract. Log. Program.* **19**(1), 27–82 (2019). <https://doi.org/10.1017/S1471068418000054>

35. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B.W. (eds.) *Handbook of Knowledge Representation, Foundations of Artificial Intelligence*, vol. 3. Elsevier, Amsterdam (2008)
36. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) *ICLP/SLP 1988*, pp. 1070–1080. MIT Press, Cambridge (1988)
37. Gonçalves, R., Knorr, M., Leite, J.: Forgetting in answer set programming - a survey. *Theory Pract. Log. Program.* **23**(1), 111–156 (2023). <https://doi.org/10.1017/S1471068421000570>
38. Heuer, J.: Automated verification of equivalence properties in advanced logic programs. Bachelor's thesis, University of Potsdam (2020). <https://arxiv.org/abs/2310.19806>
39. Heuer, J.: Automated verification of equivalence properties in advanced logic programs. In: Schwarz, S., Wenzel, M. (eds.) *WLP 2023* (2023). https://dbis.informatik.uni-halle.de/wlp2023/WLP2023_Heuer_Automated%20Verification%20of%20Equivalence%20Properties%20in%20Advanced%20Logic%20Programs.pdf
40. Huang, G.: Constructing Craig interpolation formulas. In: Du, D.-Z., Li, M. (eds.) *COCOON 1995. LNCS*, vol. 959, pp. 181–190. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0030832>
41. Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P., Schaub, T.: Clingo goes linear constraints over reals and integers. *Theory Pract. Log. Program.* **17**(5–6), 872–888 (2017). <https://doi.org/10.1017/S1471068417000242>
42. Janhunen, T., Oikarinen, E.: LPEQ and DLPEQ — translators for automated equivalence testing of logic programs. In: Lifschitz, V., Niemelä, I. (eds.) *LPNMR 2004. LNCS (LNAI)*, vol. 2923, pp. 336–340. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24609-1_30
43. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) *CAV 2013. LNCS*, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
44. Kovács, L., Voronkov, A.: First-order interpolation and interpolating proof systems. In: Eiter, T., Sands, D. (eds.) *LPAR-21. EPiC*, vol. 46, pp. 49–64. EasyChair (2017). <https://doi.org/10.29007/1qb8>
45. Lee, J., Lifschitz, V., Palla, R.: Safe formulas in the general theory of stable models (preliminary report). In: Garcia de la Banda, M., Pontelli, E. (eds.) *ICLP 2008. LNCS*, vol. 5366, pp. 672–676. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_55
46. Letz, R.: *Tableau and Connection Calculi. Structure, Complexity, Implementation*. Habilitationsschrift, TU München (1999). <http://www2.tcs.ifi.lmu.de/~letz/habil.ps>. Accessed 5 Feb 2024
47. Letz, R., Schumann, J., Bayerl, S., Bibel, W.: SETHEO: a high-performance theorem prover. *J. Autom. Reasoning* **8**(2), 183–212 (1992). <https://doi.org/10.1007/BF00244282>
48. Lifschitz, V.: Foundations of logic programming. In: Brewka, G. (ed.) *Principles of Knowledge Representation*, pp. 69–128. CSLI Publications (1996). <http://www.cs.utexas.edu/users/ai-lab?lif96b>
49. Lifschitz, V.: Thirteen definitions of a stable model. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) *Fields of Logic and Computation. LNCS*, vol. 6300, pp. 488–503. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15025-8_24
50. Lifschitz, V.: *Answer Set Programming*. Springer, Cham (2019). <https://doi.org/10.1007/978-3-030-24658-7>

51. Lifschitz, V.: Strong equivalence of logic programs with counting. *Theory Pract. Log. Program.* **22**(4), 573–588 (2022). <https://doi.org/10.1017/S1471068422000278>
52. Lifschitz, V., Lühne, P., Schaub, T.: Anthem: transforming gringo programs into first-order theories (preliminary report). *CoRR* (2018). <http://arxiv.org/abs/1810.00453>
53. Lifschitz, V., Lühne, P., Schaub, T.: Verifying strong equivalence of programs in the input language of gringo. In: Balduccini, M., Lierler, Y., Woltran, S. (eds.) *LPNMR 2019*. LNCS, vol. 11481, pp. 270–283. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20528-7_20
54. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. *ACM Trans. Comp. Log.* **2**(4), 526–541 (2001). <https://doi.org/10.1145/383779.383783>
55. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: *KR 2002*, pp. 170–176. Morgan Kaufmann (2002)
56. Lyndon, R.: An interpolation theorem in the predicate calculus. *Pac. J. Math.* **9**, 129–142 (1959)
57. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S. (eds.) *The Logic Programming Paradigm - A 25-Year Perspective*, pp. 375–398. Artificial Intelligence. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60085-2_17
58. McCune, W.: Prover9 and Mace4 (2005–2010). <http://www.cs.unm.edu/~mccune/prover9>. Accessed 5 Feb 2024
59. Nash, A., Segoufin, L., Vianu, V.: Views and queries: determinacy and rewriting. *ACM Trans. Database Syst.* **35**(3), 1–41 (2010). <https://doi.org/10.1145/1806907.1806913>
60. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* **25**(3–4), 241–273 (1999)
61. Otten, J.: Restricting backtracking in connection calculi. *AI Commun.* **23**(2–3), 159–182 (2010). <https://doi.org/10.3233/AIC-2010-0464>
62. Pearce, D., Tompits, H., Woltran, S.: Characterising equilibrium logic and nested logic programs: Reductions and complexity. *Theory Pract. Log. Program.* **9**(05), 565–616 (2009). <https://doi.org/10.1017/S147106840999010X>
63. Pearce, D., Valverde, A.: Quantified equilibrium logic and foundations for answer set programs. In: Garcia de la Banda, M., Pontelli, E. (eds.) *ICLP 2008*. LNCS, vol. 5366, pp. 546–560. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_46
64. Pearce, D., Valverde, A.: Synonymous theories and knowledge representations in answer set programming. *J. Comput. Syst. Sci.* **78**(1), 86–104 (2012). <https://doi.org/10.1016/J.JCSS.2011.02.013>
65. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) *CADE 2019*. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
66. Segoufin, L., Vianu, V.: Views and queries: determinacy and rewriting. In: *PODS 2005*, pp. 49–60. ACM (2005)
67. Slagle, J.R.: Interpolation theorems for resolution in lower predicate calculus. *JACM* **17**(3), 535–542 (1970). <https://doi.org/10.1145/321592.321604>
68. Smullyan, R.M.: *First-Order Logic*. Springer, Heidelberg (1968). Also republished with corrections by Dover publications, 1995

69. Stickel, M.E.: A Prolog technology theorem prover: implementation by an extended Prolog compiler. *J. Autom. Reasoning* **4**(4), 353–380 (1988). <https://doi.org/10.1007/BF00297245>
70. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning* **59**(4), 483–502 (2017). <https://doi.org/10.1007/s10817-017-9407-7>
71. Takeuti, G.: *Proof Theory*, 2nd edn. North-Holland, Amsterdam (1987)
72. Toman, D., Weddell, G.: *Fundamentals of Physical Design and Query Compilation*. Morgan & Claypool (2011). <https://doi.org/10.1007/978-3-031-01881-7>
73. Toman, D., Weddell, G.E.: First order rewritability in ontology-mediated querying in horn description logics. In: *AAAI 2022, IAAI 2022, EAAI 2022*, pp. 5897–5905. AAAI Press (2022). <https://doi.org/10.1609/AAAI.V36I5.20534>
74. Wernhard, C.: The PIE system for proving, interpolating and eliminating. In: Fontaine, P., Schulz, S., Urban, J. (eds.) *PAAR 2016. CEUR Workshop Proceedings*, vol. 1635, pp. 125–138. CEUR-WS.org (2016). <http://ceur-ws.org/Vol-1635/paper-11.pdf>
75. Wernhard, C.: Facets of the *PIE* environment for proving, interpolating and eliminating on the basis of first-order logic. In: Hofstedt, P., Abreu, S., John, U., Kuchen, H., Seipel, D. (eds.) *INAP/WLP/WFLP -2019. LNCS (LNAI)*, vol. 12057, pp. 160–177. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-46714-2_11
76. Wernhard, C.: Craig interpolation with clausal first-order tableaux. *J. Autom. Reasoning* **65**(5), 647–690 (2021). <https://doi.org/10.1007/s10817-021-09590-3>
77. Wernhard, C.: Range-restricted and Horn interpolation through clausal tableaux. In: Ramanayake, R., Urban, J. (eds.) *TABLEAUX 2023. LNCS (LNAI)*, vol. 14278, pp. 3–23. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-43513-3_1
78. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. *Theory Pract. Log Program* **12**(1–2), 67–96 (2012). <https://doi.org/10.1017/S1471068411000494>




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

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





Regularization in Spider-Style Strategy Discovery and Schedule Construction

Filip Bártek^{1,2} , Karel Chvalovský¹ , and Martin Suda¹ 

¹ Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Prague, Czech Republic

{[filip.bartek](mailto:filip.bartek@cvut.cz),[karel.chvalovsky](mailto:karel.chvalovsky@cvut.cz),[martin.suda](mailto:martin.suda@cvut.cz)}@cvut.cz

² Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

Abstract. To achieve the best performance, automatic theorem provers often rely on schedules of diverse proving strategies to be tried out (either sequentially or in parallel) on a given problem. In this paper, we report on a large-scale experiment with discovering strategies for the Vampire prover, targeting the FOF fragment of the TPTP library and constructing a schedule for it, based on the ideas of Andrei Voronkov’s system Spider. We examine the process from various angles, discuss the difficulty (or ease) of obtaining a strong Vampire schedule for the CASC competition, and establish how well a schedule can be expected to generalize to unseen problems and what factors influence this property.

Keywords: Saturation-based theorem proving · Proving strategies · Strategy schedule construction · Vampire

1 Introduction

In 1997 at the CADE conference, the automatic theorem prover Gandalf [30] surprised its contenders at the CASC-14 competition [29] and won the MIX division there. One of the main innovations later identified as a key to Gandalf’s success was the use of multiple theorem proving strategies executed sequentially in a time-slicing fashion [31, 32]. Nowadays, it is well accepted that a single, universal strategy of an Automatic Theorem Prover (ATP) is invariably inferior, in terms of performance, to a well-chosen portfolio of complementary strategies, most of which do not even need to be complete or very strong in isolation.

Many tools have already been designed to help theorem prover developers discover new proving strategies and/or to combine them and construct proving schedules [7, 9, 12, 16, 21, 24, 33, 34]. For example, Schäfer and Schulz employed genetic algorithms [21] for the invention of strong strategies for the E prover [23], Urban developed BliStr and used it to significantly strengthen strategies for the same prover via iterative local improvement and problem clustering [33],

Manuscript with additional appendices [1]: <https://arxiv.org/abs/2403.12869>.

© The Author(s) 2024

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 194–213, 2024.

https://doi.org/10.1007/978-3-031-63498-7_12

and, more recently, Holden and Korovin applied similar ideas in their HOS-ML system [7] to produce schedules for iProver [14]. The last work mentioned—as well as, e.g., MaLeS [16]—also include a component for *strategy selection*, the task of predicting, based on the input problem’s features, which strategy will most likely succeed on it. (Selection is an interesting topic, which is, however, orthogonal to our work and will not be further discussed here.)

For the Vampire prover [15], schedules were for a long time constructed by Andrei Voronkov using a tool called Spider, about which little was known until recently. Its author finally revealed the architectural building blocks of Spider and the ideas behind them at the Vampire Workshop 2023, declaring Spider “a secret weapon behind Vampire’s success at the CASC competitions” and “probably the most useful tool for Vampire’s support and development” [34]. Acknowledging the importance of strategies for practical ATP usability, we decided to analyze this powerful technology on our own.

In this paper, we report on a large-scale experiment with discovering strategies for Vampire, based on the ideas of Spider (recalled in Sect. 2.1).¹ We target the FOF fragment of the TPTP library [28], probably the most comprehensive benchmark set available for first-order theorem proving. As detailed in Sect. 3, we discover and evaluate (on all the FOF problems) more than 1000 targeted strategies to serve as building blocks for subsequent schedule construction.

Research on proving strategies is sometimes frowned upon as mere “tuning for competitions”. While we briefly pause to discuss this aspect in Sect. 4, our main interest in this work is to establish how well a schedule can be expected to generalize to unseen problems. For this purpose, we adopt the standard practice from statistics to randomly split the available problems into a train set and a test set, construct a schedule on one, and evaluate it on the other. In Sect. 6, we then identify several techniques that *regularize*, i.e., have the tendency to improve the test performance while possibly sacrificing the training one.

Optimal schedule construction under some time budget can be expressed as a mixed integer program and solved (given enough time) using a dedicated tool [7, 24]. Here, we propose to instead use a simple heuristic from the related set cover problem [3], which leads to a polynomial-time greedy algorithm (Sect. 5). The algorithm maintains the important ability to assign different time limits to different strategies, is much faster than optimal solving (which may overfit to the train set in some scenarios), allows for easy experimentation with regularization techniques, and, in a certain sense made precise later, does not require committing to a single predetermined time budget.

In summary, we make the following main contributions:

- We outline a pragmatic approach to schedule construction that uses a greedy algorithm (Sect. 5), contrasting it with optimal schedules in terms of the quality of the schedules and the computational resources required for their construction (Sect. 6.2). In particular, our findings demonstrate a relative efficacy of the greedy approach for datasets similar to our own.

¹ Not claiming any credit for these, potential errors in the explanation are ours alone.

- Leveraging the adaptability of the greedy algorithm, we introduce a range of regularization techniques aimed at improving the robustness of the schedules in unseen data (Sect. 6). To the best of our knowledge, this represents the first systematic exploration into regularization of strategy schedules.
- The strategy discovery and evaluation is a computationally expensive process, which in our case took more than twenty days on 120 CPU cores. At the same time, there are further interesting questions concerning Vampire’s strategies than we could answer in this work. To facilitate research on this paper’s topic, we made the corresponding data set available online [2].

2 Preliminaries

The behavior of Vampire is controlled by approximately one hundred *options*. These options configure the preprocessing and classification steps, control the saturation algorithm, clause and literal selection heuristics, determine the choice of generating inferences as well as redundancy elimination and simplification rules, and more. Most of these options range over the Boolean or a small finite domain, a few are numeric (integer or float), and several represent ratios.

Every option has a *default* value, which is typically the most universally useful one. Some option settings make Vampire incomplete. This is automatically recognized, so that when the prover finitely saturates the input without discovering a contradiction, it will report “unknown” (rather than “satisfiable”).

A *strategy* is determined by specifying the values of all options. A *schedule* is a sequence $(s_i, t_i)_{i=1}^n$ of strategies s_i together with assigned time limits t_i , intended to be executed in the prescribed order. We stress that in this work we do not consider schedules that would branch depending on problem features.

2.1 Spider-Style Strategy Discovery and Schedule Construction

We are given a set of problems P and a prover with its space of available strategies \mathcal{S} . *Strategy discovery* and *schedule construction* are two separate phases. We work under the premise that the larger and more diverse a set of strategies we first collect, the better for later constructing a good schedule.

Strategy discovery consists of three stages: random probing, strategy optimization, and evaluation, which can be repeated as long as progress is made.

Random Probing. We start strategy discovery with an empty pool of strategies $S = \emptyset$. A straightforward way to make sure that a new strategy substantially contributes to the current pool S is to always try to solve a problem not yet solved (or *covered*) by any strategy collected so far. We repeatedly pick such a problem and try to solve it using a *randomly sampled* strategy out of the totality of all available strategies \mathcal{S} . The sampling distribution may be adapted to prefer option values that were successful in the past (cf. Sect. 3.3). This stage is computationally demanding, but can be massively parallelized.

Strategy Optimization. Each newly discovered strategy s , solving an as-of-yet uncovered problem p , will get *optimized* to be as fast as possible at solving p . One explores the strategy neighborhood by iterating over the options (possibly in several rounds), varying option values, and committing to changes that lead to a (local) improvement in terms of solution time or, as a tie-breaker, to a default option value where time differences seem negligible. We evaluate the impact of this stage in Sect. 3.4.

Strategy Evaluation. In the final stage of the discovery process, having obtained an optimized version s' of s , we evaluate s' on all our problems P . (This is another computationally expensive, but parallelizable step.) Thus, we enrich our pool and update our statistics about covered problems. Note that every strategy s' we obtain this way is associated with the problem $p_{s'}$ for which it was originally discovered. We will call this problem the *witness problem* of s' .

Schedule Construction can be tried as soon as a sufficiently rich (cf. Sect. 3) pool of strategies is collected. Since we, for every collected strategy, know how it behaves on each problem, we can pose schedule construction as an optimization task to be solved, e.g., by a (mixed) integer programming (MIP) solver.

In more detail: We seek to allocate time slices $t_s > 0$ to some of the strategies $s \in S$ to cover as many problems as possible while remaining in sum below a given time budget T [7, 24]. Alternatively, we may try to cover all the problems known to be solvable in as little total time as possible.² In this paper, we describe an alternative schedule construction method based on a greedy heuristic, with a polynomial running time guarantee and other favorable properties (Sect. 5).

2.2 CPU Instructions as a Measure of Time

We will measure computation time in terms of the number of user instructions executed (as available on Linux systems through the `perf` tool). This is, in our experience, more precise and more stable (on architectures with many cores and many concurrently running processes) than measuring real time.³

In fact, we report *megainstructions* (Mi), where 1 $Mi = 2^{20}$ instructions reported by `perf`. On contemporary hardware, 2000 Mi will typically get used up in a bit less than a second and 256 000 Mi in around 2 min of CPU time. We also set 1 Mi as the granularity for the time limits in our schedules.

3 Strategy Discovery Experiment

Following the recipe outlined in Sect. 2.1, we set out to collect a pool of Vampire (version 4.8) strategies covering the first-order form (FOF) fragment of the

² Strictly speaking, these only give us a set of strategy-time pairs (as opposed to a sequence). However, the strategies can be ordered heuristically afterward.

³ For a more thorough motivation, see Appendix A of [26].

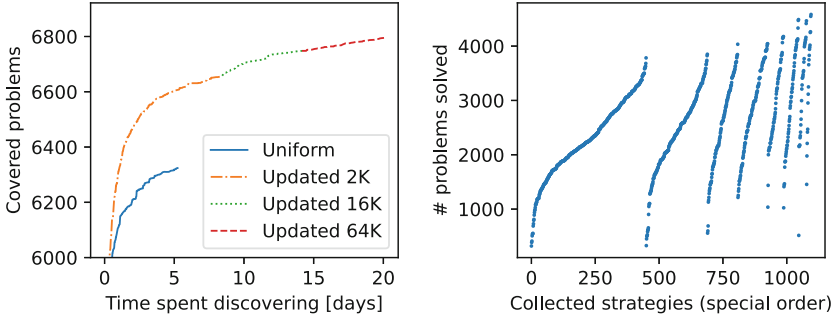


Fig. 1. Strategy discovery. *Left:* problem coverage growth in time (uniform strategy sampling distribution vs. an updated one). *Right:* collected strategies ordered by limit (2000, 4000, . . . , 256 000 Mi) and, secondarily, by how many problems can each solve.

TPTP library [28] version 8.2.0. We focused only on proving, so left out all the problems known to be satisfiable, which left us with a set P of 7866 problems. Parallelizing the process where possible, we strived to fully utilize 120 cores (AMD EPYC 7513, 3.6 GHz) of our server equipped with 500 GB RAM.

We let the process run for a total of 20.1 days, in the end covering 6796 problems, as plotted in Fig. 1 (left). The effect of diminishing returns is clearly visible; however, we cannot claim we have exhausted all the possibilities. In the last day alone, 8 strategies were added and 9 new problems were solved.

The rest of Fig. 1 is gradually explained in the following as we cover some important details regarding the strategy discovery process.

3.1 Initial Strategy and Varying Instruction Limits

We seeded the pool of strategies by first evaluating Vampire’s default strategy for the maximum considered time limit of 256 000 Mi, solving 4264 problems out of the total 7866.

To save computation time, we did not probe or evaluate all subsequent strategies for this maximum limit. Instead, to exponentially prefer low limits to high ones, we made use of the Luby sequence⁴ [18] known for its utility in the restart strategies of modern SAT solvers. Our own use of the sequence was as follows.

The lowest limit was initially set to 2000 Mi and, multiplying the Luby sequence members by this number, we got the progression 2000, 2000, 4000, 2000, 2000, 4000, 8000, . . . as the prescribed limits for subsequent probe iterations. This sequence reaches 256 000 Mi for the first time in 255 steps. At that point, we stopped following the Luby sequence and instead started from the beginning (to avoid eventually reaching limits higher than 256 000 Mi).

After four such cycles, the lowest, that is 2000 Mi, limit probes stopped producing new solutions (a sampling timeout of 1 h per iteration was imposed).

⁴ <https://oeis.org/A182105>.

Here, after almost 8.5 d, the “updated 2K” plot ends in Fig. 1 (left). We then increased the lowest limit to 16 000 Mi and continued in an analogous fashion for 155 iterations and 5.7 more days (“updated 16K”) and eventually increased the lowest limit to 64 000 Mi (“updated 64K”) until the end.

Figure 1 (right) is a scatter plot showing the totality of 1096 strategies that we finally obtained and how they individually perform. The primary order on the x axis is by the limit and allows us to make a rough comparison of the number of strategies in each limit group (2000 Mi, 4000 Mi, . . . , 256 000 Mi, from left to right). It is also clear that many strategies (across the limit groups) are, in terms of problem coverage, individually very weak, yet each at some point contributed to solving a problem considered (at that point) challenging.

3.2 Problem Sampling

While the guiding principle of random probing is to constantly aim for solving an as-of-yet unsolved problem, we modified this criterion slightly to produce a set of strategies better suited for an unbiased estimation of schedule performance on unseen problems (as detailed in the second half of this paper).

Namely, in each iteration i , we “forgot” a random half P_i^F of all problems P , considered only those strategies (discovered so far) whose witness problem lies in the remaining half $P_i^R = P \setminus P_i^F$, and aimed for solving a random problem in P_i^R not covered by any of these strategies. This likely slowed the overall growth of coverage, as many problems would need to be covered several times due to the changing perspective of P_i^R . However, we got a (probabilistic) guarantee that any (not too small) subset $P' \subseteq P$ will contain enough witness problems such that their corresponding strategies will cover P' well.

3.3 Strategy Sampling

We sampled a random strategy by independently choosing a random value for each option. The only exception were dependent options. For example, it does not make sense to configure the AVATAR architecture (changing options such as `acc`, which enables congruence closure under AVATAR) if the main AVATAR option (`av`) is set to `off`. Such complications can be easily avoided by following, during the sampling, a topological order that respects the option dependencies. (For example, we sample `acc` only after the value `on` has been chosen for `av`.)

Even under the assumption of option independence, the mean time in which a random strategy solves a new problem can be strongly influenced by the value distributions for each option. This is because some option values are rarely useful and may even substantially reduce the prover performance, for example, if they lead to a highly incomplete strategy.⁵ Nevertheless, not to preclude the

⁵ An extreme example is turning off *binary resolution*, the main inference for non-equational reasoning. This can still be useful, for instance when replaced by *unit resulting resolution* [15], but our sampling needs to discover this by chance.

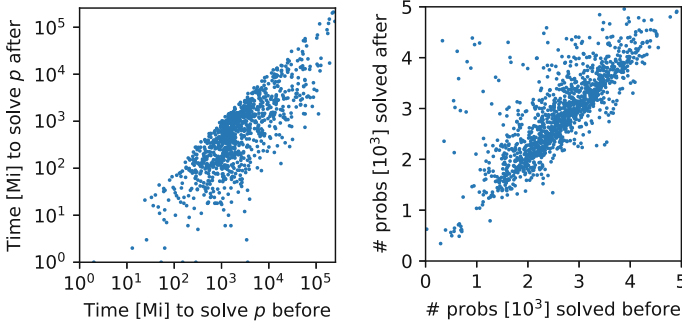


Fig. 2. Strategy optimization scatter plots. *Left:* time needed to solve strategy’s witness problem (a log–log plot). *Right:* the total number of problems (in thousands) solved.

possibility of discovering arbitrarily wild strategies, we initially sampled every option uniformly where possible.⁶

Once we collected enough strategies,⁷ we updated the frequencies for sampling finite-domain options (which make up the majority of all options) by counting how many times each value occurred in a strategy that, at the moment of its discovery, solved a previously unsolved problem. (This was done before a strategy got optimized. Otherwise the frequencies would be skewed toward the default, especially for option values that rarely help but almost never hurt.)

The effect of using an updated sampling distribution for strategy discovery can be seen in Fig. 1 (left). We ran two independent versions of the discovery process, one with the uniform distribution and one with the updated distribution. We abandoned the uniform one after approximately 5 d, by which time it had covered 6324 problems compared to 6607 covered with the help of the updated distribution at the same mark. We can see that the rate at which we were able to solve new problems became substantially higher with the updated distribution.

3.4 Impact of Strategy Optimization

Once random probing finds a new strategy s that solves a new problem p , the task of optimization (recall Sect. 2.1) is to search the option-value neighborhood of s for a strategy s' that solves p in as few instructions as possible and preferably uses default option values (where this does not compromise performance on p).

The impact of optimization is demonstrated in Fig. 2. On the left, we can see that, almost invariably, optimization substantially improves the performance of the discovered strategy on its witness problem p . The geometric mean of the improvement ratio we observed was 4.2 (and the median 3.2). The right

⁶ Exceptions were: 1. The ratios: e.g., for `age_weight_ratio` we sampled uniformly its binary logarithm (in the range -10 and 4) and turned this into a ratio afterward (thus getting values between $1 : 1024$ and $16 : 1$); 2. Unbounded integers (an example being the *naming threshold* [20]), for which we used a geometric distribution instead.

⁷ This was done in an earlier version of the main experiment.

scatter plot shows the overall performance of each strategy.⁸ Here, the observed improvement is $\times 1.09$ on average (median 1.03), and the improvement is solely an effect of setting option values to default where possible (without this feature, we would get a geometric mean of the improvement 0.84 and median 0.91). In this sense, the tendency to pick default values regularizes the strategies, making them more powerful also on problems other than their witness problem.

3.5 Parsing Does Not Count

When collecting the performance data about the strategies, we decided to ignore the time it takes Vampire to parse the input problem. This was also reflected in the instruction limiting, so that running Vampire with a limit of, e.g., 2000 Mi would allow a problem to be solved if it takes at most 2000 Mi on top of what is necessary to parse the problem.

The main reason for this decision is that Vampire, in its strategy scheduling mode, starts dispatching strategies only after having parsed the problem, which is done only once. Thus, from the perspective of individual strategies, parsing time is a form of a sunk cost, something that has already been paid.

Although more complex approaches to taking parse time into account when optimizing schedules are possible, we in this work simply pretend that problem parsing always costs 0 instructions. This should be taken into account when interpreting our simulated performance results reported next (in Sect. 4, but also in Sect. 6.2).

4 One Schedule to Cover Them All

Having collected our strategies, let us pretend that we already know how to construct a schedule (to be detailed in Sect. 5) and use this ability to answer some imminent questions, most notably: How much can we now benefit?

Figure 3 plots the cumulative performance (a.k.a. “cactus plot”) of schedules we could build after 2 h, 6 h, 1 day, and full 20.1 days of the strategy discovery. The dashed vertical line denotes the time limit of 256 000 Mi, which roughly corresponds to a 2-minute prover run. For reference, we also plot the behavior of Vampire’s default strategy. We can see that already after two hours of strategy discovery, we could construct a schedule improving on the default strategy by 26% (from 4264 to 5403 problems solved). Although the added value per hour spent searching gradually drops, the 20.1 days schedule is still 4% better than the 1 day one (improving from 6197 to 6449 at 256 000 Mi).

The plot’s x -axis ends at $8 \cdot 256\,000$ Mi, which roughly corresponds to the time limit used by the most recent CASC competitions [27] in the FOF division (i.e., 2 min on 8 cores). The strongest schedule shown in the figure manages to

⁸ In a previous version of the main experiment, we evaluated each strategy both before and after optimization, which gave rise to this plot.

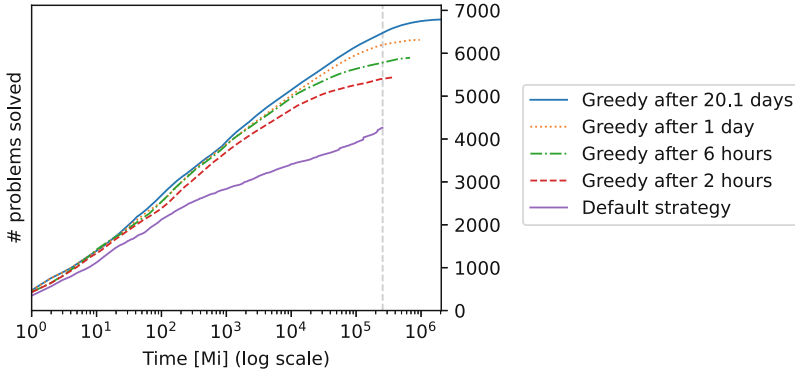


Fig. 3. Cumulative performance of several greedy schedules, each using a subset of the discovered strategies as gathered in time, compared with Vampire’s default strategy

solve 6789 problems (of the 6796 covered in total) at that mark.⁹ We remark that this schedule, in the end, employs only 577 of the 1096 available strategies, which points towards a noticeable redundancy in the strategy discovery process.

One way to fit all the solvable problems below the CASC budget would be to use a standard trick and split the totality of problems P into two or more easy-to-define syntactic classes (e.g., Horn problems, problems with equality, large problems, etc.) and construct dedicated schedules for each class in isolation. The prover can then be dispatched to run an appropriate schedule once the input problem features are read. We do not explore this option here. Intuitively, by splitting P into smaller subsets, the risk of overfitting to just the problems for which the strategies were discovered increases, and we mainly want to explore here the opposite, the ability of a schedule to generalize to unseen problems.

5 Greedy Schedule Construction

Having collected a set of strategies S and evaluated each on the problems in P , let us by $E_p^s : S \times P \rightarrow \mathbb{N} \cup \{\infty\}$ denote the *evaluation matrix*, which records the obtained solutions times (and uses ∞ to signify a failure to solve a problem within the evaluation time limit used). Given a time budget T , the *schedule construction problem* (SCP) is the task of assigning a time limit $t_s \in \mathbb{N}$ to every strategy $s \in S$, such that the number of covered problems

$$\left| \bigcup_{s \in S} \{p \in P \mid E_p^s \leq t_s\} \right|,$$

subject to the constraint $\sum_{s \in S} t_s \leq T$, is maximized.

⁹ It is possible to solve all 6796 covered problems with a schedule that spans 2 582 228 Mi. This is the optimum length – no shorter schedule solving all covered problems exists.

Algorithm 1. Greedy schedule construction (base version)

Input: Problems P , strategies S , solution times E_p^s , time budget T
Output: Pre-schedule $t_s : S \rightarrow \mathbb{N}$

```

1:  $t_s \leftarrow \mathbf{0}$  ▷ Start with zeros everywhere
2:  $T' \leftarrow T, P' \leftarrow P$  ▷ Remaining budget, remaining problems
3: while the maximum just below is positive do
4:    $s, t \leftarrow \operatorname{argmax}_{s \in S, t \in \mathbb{N}, 0 < (t - t_s) \leq T'} |\{p \in P' \mid E_p^s \leq t\}| / (t - t_s)$ 
5:    $T' \leftarrow T' - (t - t_s)$  ▷ Update the remaining budget
6:    $t_s \leftarrow t$  ▷ Extend the pre-schedule
7:    $P' \leftarrow \{p \in P' \mid E_p^s > t\}$  ▷ Remove covered problems from  $P'$ 

```

To obtain a schedule as a sequence (as defined in Sect. 2), we would need to order the strategies having $t_s > 0$. This can, in practice, be done in various ways, but since the order does not influence the predicted performance of the schedule under the budget T , we keep it here unspecified (and refer to the mere time assignment t_s as a *pre-schedule* where the distinction matters).

Although it is straightforward to encode SCP as a mixed integer program and attempt to solve it exactly (though it is an NP-hard problem), an adaptation of a greedy heuristic from a closely related (budgeted) maximum coverage problem [3, 13] works surprisingly well in practice and runs in time polynomial in the size of E_p^s . The key idea is to greedily maximize the number of newly covered problems *divided* by the amount of time this additionally requires.

Algorithm 1 shows the corresponding pseudocode. It starts from an empty schedule t_s and iteratively extends it in a greedy fashion. The key criterion appears on line 4. Note that this line corresponds to an iteration over all available strategies S and, for each strategy s , all meaningful time limits (which are only those where a new problem gets solved by s , so their number is bounded by $|P|$).

Algorithm 1 departs from the obvious adaptation of the above-mentioned greedy algorithm for the set covering problem [3] in that we allow extending a slice of a strategy s that is already included in the schedule (that is, has $t_s > 0$) and “charge the extension” only for the additional time it claims (i.e., $t - t_s$). This *slice extension* trick turns out to be important for good performance.¹⁰

5.1 Do We Need a Budget?

A budget-less version of Algorithm 1 is easy to obtain (imagine T being very large). When running on real-world E_p^s (from evaluated Vampire strategies), we noticed that the length of a typical extension ($t - t_s$) tends to be small relative to the current used-up time $\sum_{s \in S} t_s$ and that the presence of a budget starts affecting the result only when the used-up time comes close to the budget.

As a consequence, if we run a budget-less version and, after each iteration, record the pair $(\sum_{s \in S} t_s, |P \setminus P'|)$, we get a good estimate (in a single run) of how the algorithm would perform for a whole (densely inhabited) sequence of

¹⁰ The degree of importance of slice extension can be observed in Fig. 5 in Appendix A.

relevant budgets. This is how the plot in Fig. 3 was obtained. Note that this would be prohibitively expensive to do when trying to solve the SCP optimally.

We can also use this observation in an actual prover. If we record and store a journal of the budget-less run, remembering which strategy got extended in each iteration and by how much, we can, given a concrete budget T , quickly replay the journal just to the point before filling up T , and thus get a good schedule for the budget T without having to optimize specifically for T .

6 Regularization in Schedule Construction

To estimate the future performance of a constructed schedule on previously unseen problems, we adopt the standard methodology used in statistics, randomly split our problem set P into a train set P_{train} and a test set P_{test} , construct a schedule for the first, and evaluate it on the second.

To reduce the variance in the estimate, we use many such random splits and average the results. In the experiments reported in the following, we actually compute an average over several rounds of 5-fold cross-validation [6]. This means that the size of P_{train} is always 80.0 % and the size P_{test} 20.0% of our problem set P . However, we *re-scale* the reported train and test performance back to the size of the whole problem set P to express them in units that are immediately comparable. We note that the reported performance is obtained though simulation, i.e., it is only based on the evaluation matrix E_p^s .

Training Strategy Sets. We retroactively simulate the effect of discovering strategies only for current training problems P_{train} . Given our collected pool of strategies S , we obtain the training strategy set S_{train} by excluding those strategies from S whose witness problem lies outside P_{train} (cf. Sect. 3.2). When a schedule is optimized on the problem set P_{train} , the training data consists of the results of the evaluations of strategies S_{train} on problems P_{train} .

6.1 Regularization Methods

We propose several modifications of greedy schedule construction (Algorithm 1) with the aim of improving its performance on unseen problems (the test set performance) while possibly sacrificing some of its training performance.

With the base version, we observed that it could often solve more test problems by assigning more time to strategies introduced into the schedule in early iterations, at the expense of strategies added later (the latter presumably covering just a few expensive training problems and being over-specialized to them). Most of the modifications described next assign more time to strategies added during early iterations, each according to a different heuristic.

Slack. The most straightforward regularization we explored extends each non-zero strategy time limit t_s in the schedule by multiplying it by the multiplicative slack $w \geq 1$ and adding the additive slack $b \in \{0, 1, \dots\}$. For each

$t_s > 0$, the new limit t'_s is therefore $t_s \cdot w + b$. To avoid overshooting the budget, we keep track of the total length of the extended schedule during the construction (implementation details are slightly more complicated but not immediately important). The parameters w and b control the degree of regularization, and with $w = 1$ and $b = 0$, we get the base algorithm.

Temporal Reward Adjustment. In each iteration of the base greedy algorithm, we select a combination of strategy s and time limit t that maximizes the number of newly solved problems n per time t . Intuitively, the relative degree to which these two quantities influence the selection is arbitrary. To allow stressing n more or less with respect to t , we exponentiate n by a regularization parameter $\alpha \geq 0$, so the decision criterion becomes $\frac{n^\alpha}{t}$. For small values of α , the algorithm values the time more and becomes eager to solve problems early. For large values of α , on the other hand, the algorithm values the problems more and prefers longer slices that cover more problems. For example, for $\alpha = 1.5$, the algorithm prefers solving 2 problems in 5000 Mi to solving 1 problem in 2000 Mi. Compare this to $\alpha = 1$ (the base algorithm), which would rank these slices the other way around.

Diminishing Problem Rewards. By covering a training problem with more than one strategy, we cover it robustly: When a similar testing problem is solved by only one of these strategies, the schedule still manages to solve it. However, the base greedy algorithm does not strive to cover any problem more than once: as soon as a problem is covered by one strategy, this problem stops participating in the scheduling criterion. This is the case even when covering the problem again would cost relatively little time.

Regularization by diminishing problem rewards covers problems robustly by rewarding strategy s not only by the number of *new* problems it covers but also by the problems covered by s that are already covered by the schedule. This is achieved by modifying the slice selection criterion. Instead of maximizing the number of new problems solved per time, we maximize the total reward per time, which is defined as follows: Each problem contributes the reward β^k , where k is the number of times the schedule has covered the problem and β is a regularization parameter ($0 \leq \beta \leq 1$). We define $0^0 = 1$ so that $\beta = 0$ preserves the original behavior of the base algorithm.

For example, for $\beta = 0.1$, each problem contributes the reward 1 the first time it is covered, 0.1 the second time, 0.01 the third time, etc. Informally, the algorithm values covering a problem the second time in time t as much as covering a new problem in time $10 \cdot t$.

These modifications are independent and can be arbitrarily combined.

6.2 Experimental Results

We evaluated the behavior of the previously proposed techniques using three time budgets: 16 000 Mi (≈ 8 s), 64 000 Mi (≈ 32 s), and 256 000 Mi (≈ 2 min).

Optimal Schedule Constructor. In the existing approaches to the construction of strategy schedules [7, 24], it is common to encode the SCP (see Sect. 5) as a mixed-integer program and use a MIP solver to find an exact solution. We implemented such an optimal schedule construction (OSC) by encoding the problem¹¹ in Gurobi [5] (ver. 10.0.3) and compared OSC to the base greedy schedule construction (Algorithm 1) on 10 random 80 : 20 splits.

For the budget of 256 000 Mi, it takes Gurobi over 16 h to find an optimal schedule, whereas the greedy algorithm finds a schedule in less than a minute. The optimal schedule solves, on average, 45.0 (resp. 8.5) more problems than the greedy schedule on P_{train} (resp. on P_{test}) when re-scaled to $|P|$. For the 16 000 Mi and 64 000 Mi budgets, Gurobi does not solve the optimal schedule within a reasonable time limit. For example, after 24 h, the relative gaps between the lower and upper objective bound are 5.38 % and 1.43 %, respectively. This makes the OSC impractical to use as a baseline for our regularization experiments.¹²

Regularization of the Greedy Algorithm. To estimate the performance of the proposed regularization methods, we evaluated each variant on 50 random splits (10 times 5-fold cross-validation). We assessed the algorithm’s response to each regularization parameter in isolation. For each parameter, we evaluated regularly spaced values from a promising interval covering the default value ($b = 0$, $w = 1$, $\alpha = 1$, $\beta = 0$). Figure 4 demonstrates the effect of these variations on the train and test performance for the budget 64 000 Mi.¹³

Temporal reward adjustment was the most powerful of the regularizations, improving test performance for all the evaluated values of α between 1.1 and 2.0. Surprisingly, the values 1.1 and 1.2 also improved the train performance, suggesting that the default greedy algorithm is too time-aggressive on our dataset.

Table 1 compares the performance of notable configurations of the greedy algorithm. Specifically, we include evaluations of the base greedy algorithm and the best of the evaluated parameter values for each of the regularizations. The table also illustrates the effect of regularizations on the computational time of the greedy schedule construction: $\beta > 0$ slows the procedure down and $\alpha > 1$ speeds it up.

In a subsequent experiment, we searched for a strong combination of regularizations by local search from the strongest single-parameter regularization ($\alpha = 1.7$). This yielded a negligible improvement over $\alpha = 1.7$: The best observed test performance was 5707 ($\alpha = 1.7$ and $b = 30$), compared to 5704 of $\alpha = 1.7$.

Finally, we briefly explored the interactions between the budget and the optimal values of the regularization parameters. For each of the three budgets of interest and each of the regularization parameters, we identified the best param-

¹¹ We used a straightforward encoding similar to the encoding described by Holden and Korovin [7].

¹² A better encoding and solver settings may improve on this. However, we suspect the problem to be hard; we tried some modifications with similar (or worse) results.

¹³ This budget seems to be the most practically relevant (e.g., for the application in interactive theorem provers). The other two budgets are detailed in Appendix A.

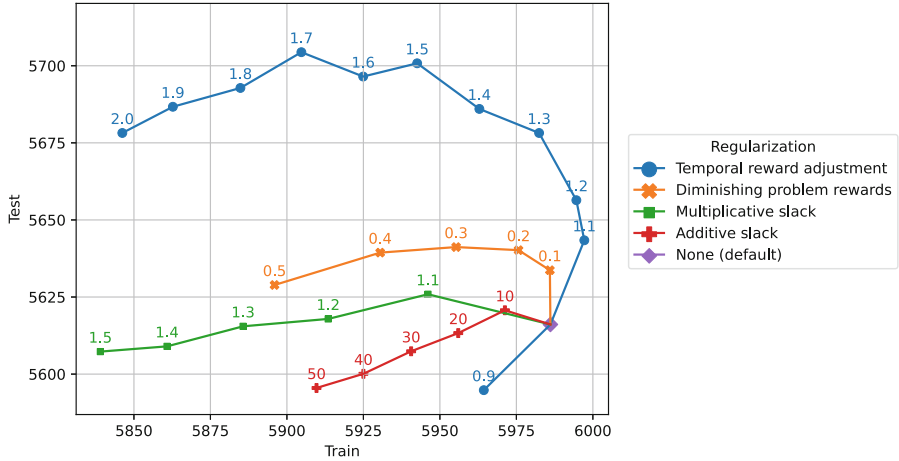


Fig. 4. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budget 64 000. Performance is the mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

eter value from the evaluation grid. Table 2 shows that the best configurations of all the regularizations except multiplicative slack vary across budgets.¹⁴

7 Related Work

Outside the realm of theorem proving, strategy discovery belongs to the topic of *algorithm configuration* [22], where the task is to look for a strong configuration of a parameterized algorithm automatically. Prominent general-purpose algorithm configuration procedures include ParamILS [8] and SMAC [17].

To gather a portfolio of complementary configurations, Hydra [36] searches for them in rounds, trying to maximize the marginal contribution against all the configurations identified previously. Cedalion [25] is interesting in that it maximizes such contribution *per unit time*, similarly to our heuristic for greedy schedule construction. Both have in common that they, a priori, consider all the input problems in their criterion. BliStr and related approaches [7, 9, 11, 12, 33], on the other hand, combine strategy improvement with problem clustering to breed strategies that are “local experts” on similar problems. Spider [34] is even more radical in this direction and optimizes each strategy on a single problem.¹⁵

¹⁴ See a more detailed comparison in Appendix A.

¹⁵ Although the preference for default option values as a secondary criterion, at the same time, helps to push for good general performance (see Sect. 3.4).

Table 1. Comparison of regularizations of the greedy schedule construction algorithm for the budget 64 000 Mi. Performance is the mean number of problems solved out of 7866 across 50 splits. Time to fit is the mean time to construct a schedule in seconds.

Regularization	Performance		Time to fit [s]
	Test	Train	
$\alpha = 1.7$	5704	5905	4
$\beta = 0.3$	5641	5955	67
$w = 1.1$	5626	5946	19
$b = 10$	5621	5971	20
None (default)	5616	5986	20

Table 2. Best observed values of regularization parameters for various budgets

Budget [Mi]	Best parameter value			
	b	w	α	β
16000	0	1.1	1.3	0.2
64000	10	1.1	1.7	0.3
256000	20	1.1	1.4	0.2

Once a portfolio of strategies is known, it may be used in one of several ways to solve a new input problem: execute all strategies in parallel [36], select a single strategy [9], select one of pre-computed schedules [7], construct a custom strategy schedule [19], schedule strategies dynamically [16], or use a pre-computed static schedule [12, 24]. The latter is the approach we explored in this work.

A popular approach to construct a static schedule (besides solving SCP optimally [7, 24]) is to greedily stack uniformly-timed slices [12].¹⁶ Regularization in this context is discussed by Jakubuv et al. [10]. Finally, a different greedy approach to schedule construction was already proposed in *p-SETHEO* [35].

8 Conclusion

In this work, we conducted an independent evaluation of Spider-style [34] strategy discovery and schedule creation. Focusing on the FOF fragment of the TPTP library, we collected over a thousand Vampire proving strategies, each a priori optimized to perform well on a single problem. Using these strategies, it is easy to construct a single monolithic schedule which covers most of the problems

¹⁶ However, uniformly-timed slices only get close to the performance of our greedy schedule at a small region depending on the slice time limit used.

known to be solvable within the budget used by the CASC competition. This suggests that for CASC not to be mainly a competition in memorization, using a substantial set of previously unseen problems each year is essential.

To construct strong schedules using the discovered strategies, we proposed a greedy schedule construction procedure, which can compete with optimal approaches. For a time budget of approximately 2 min, the greedy algorithm takes less than a minute to produce a schedule that solves more than 99.0% as many problems as an optimal schedule, which takes more than 16 h to generate. For shorter time budgets, optimal schedule construction is no longer feasible, while greedy construction still produces relatively strong schedules.

This surprising strength of the greedy scheduler can be further reinforced by various regularization mechanisms, which constitute the main contribution of this work. An appropriately chosen regularization allows us to outperform the optimal schedule on unseen problems. Finally, the runtime speed and simplicity of the greedy schedule construction algorithm and the regularization techniques make them attractive for reuse and further experimentation.

Acknowledgment. This work was supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the project RICAIP no. 857306 under the EU-H2020 programme, the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37, and the Czech Science Foundation project no. 24-12759S.

A Experimental Results on Various Budgets

In addition to the budget of 64 000 Mi (approx 32 s), which we discussed in Sect. 6.2, we evaluated the schedule construction algorithms on the budgets of 16 000 Mi (approx. 8 s) and 256 000 Mi (approx. 2 min). Figure 5 shows the results of these evaluations. It shows namely that temporal reward adjustment is the most powerful of the regularizations under consideration for all of these budgets, and that the optimal values of most of the regularization parameters vary across budgets.

To demonstrate the effect of the slice extension trick described in Sect. 5, we also include two weaker versions of the base greedy algorithm: one without slice extension and one with the slice extension restricted to the most recent slice (“conservative slice extension”). Both of these modifications allow including any single strategy in the schedule more than once, which is implemented in a straightforward fashion.

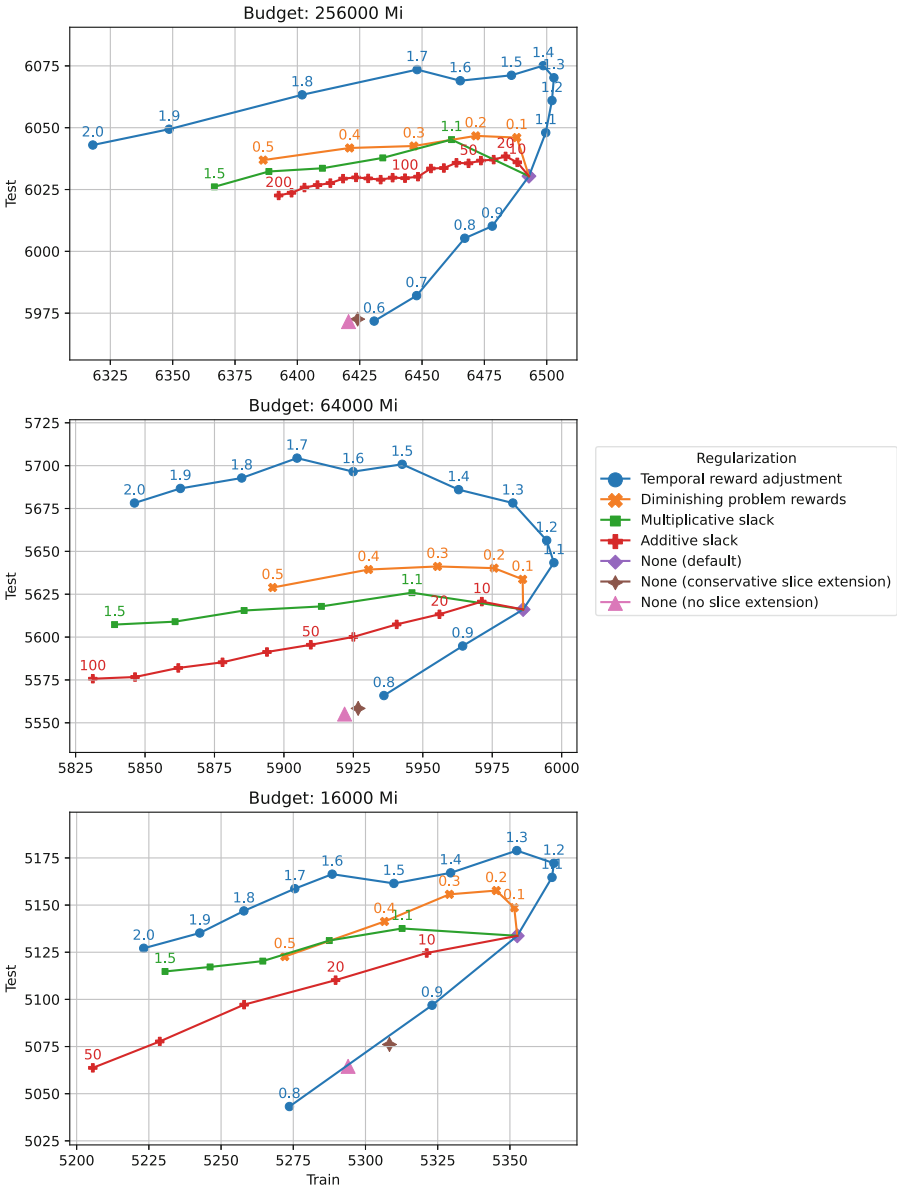


Fig. 5. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budgets 256 000 Mi (*top*), 64 000 Mi (*middle*), and 16 000 Mi (*bottom*). Performance is mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

References

1. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction (2024). <https://doi.org/10.48550/arXiv.2403.12869>
2. Bártek, F., Suda, M.: Vampire strategy performance measurements (2024). <https://doi.org/10.5281/zenodo.10814478>
3. Chvátal, V.: A greedy heuristic for the set-covering problem. *Math. Oper. Res.* **4**(3), 233–235 (1979). <https://doi.org/10.1287/moor.4.3.233>
4. Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.): Global Conference on Artificial Intelligence, GCAI 2015, Tbilis, 16–19 October 2015, EPiC Series in Computing, vol. 36. EasyChair (2015). <https://easychair.org/publications/volume/GCAI.2015>
5. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual (2023). <https://www.gurobi.com>
6. Hastie, T., Tibshirani, R., Friedman, J.: *The Elements of Statistical Learning*, 2nd edn. SSS, Springer, New York (2009). <https://doi.org/10.1007/978-0-387-84858-7>
7. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Coen, C.S. (eds.) *CICM 2021*. LNCS, vol. 12833, pp. 107–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81097-9_8
8. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. *J. Artif. Intell. Res.* **36**, 267–306 (2009). <https://doi.org/10.1613/JAIR.2861>
9. Hula, J., Jakubuv, J., Janota, M., Kubej, L.: Targeted configuration of an SMT solver. In: Buzzard, K., Kutsia, T. (eds.) *CICM 2022*. LNCS, vol. 13467, pp. 256–271. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16681-5_18
10. Jakubuv, J., et al.: MizAR 60 for Mizar 50. In: Naumowicz, A., Thiemann, R. (eds.) *14th International Conference on Interactive Theorem Proving, ITP 2023*, 31 July to 4 August 2023, Białystok. *LIPICs*, vol. 268, pp. 19:1–19:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICs.ITP.2023.19>
11. Jakubuv, J., Suda, M., Urban, J.: Automated invention of strategies and term orderings for Vampire. In: Benzmüller, C., Lisetti, C.L., Theobald, M. (eds.) *GCAI 2017*, 3rd Global Conference on Artificial Intelligence, Miami, 18–22 October 2017. *EPiC Series in Computing*, vol. 50, pp. 121–133. EasyChair (2017). <https://doi.org/10.29007/XGHJ>
12. Jakubuv, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: Bertot, Y., Vafeiadis, V. (eds.) *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, Paris, 16–17 January 2017, pp. 43–52. ACM (2017). <https://doi.org/10.1145/3018610.3018619>
13. Khuller, S., Moss, A., Naor, J.: The budgeted maximum coverage problem. *Inf. Process. Lett.* **70**(1), 39–45 (1999). [https://doi.org/10.1016/S0020-0190\(99\)00031-9](https://doi.org/10.1016/S0020-0190(99)00031-9)
14. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) *IJCAR 2008*. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
15. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1

16. Kühlwein, D., Urban, J.: MaLeS: a framework for automatic tuning of automated theorem provers. *J. Autom. Reason.* **55**(2), 91–116 (2015). <https://doi.org/10.1007/s10817-015-9329-1>
17. Lindauer, M., et al.: SMAC3: a versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.* **23**, 54:1–54:9 (2022). <http://jmlr.org/papers/v23/21-0888.html>
18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.* **47**(4), 173–180 (1993). [https://doi.org/10.1016/0020-0190\(93\)90029-9](https://doi.org/10.1016/0020-0190(93)90029-9)
19. Mangla, C., Holden, S.B., Paulson, L.C.: Bayesian ranking for strategy scheduling in automated theorem provers. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) *IJCAR 2022*. LNCS, vol. 13385, pp. 559–577. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_33
20. Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, 19 September–2 October 2016, Berlin. *EPiC Series in Computing*, vol. 41, pp. 11–23. EasyChair (2016). <https://doi.org/10.29007/DZfZ>
21. Schäfer, S., Schulz, S.: Breeding theorem proving heuristics with genetic algorithms. In: Gottlob et al. [4], pp. 263–274. <https://doi.org/10.29007/gms9>
22. Schede, E., et al.: A survey of methods for automated algorithm configuration. *J. Artif. Intell. Res.* **75**, 425–487 (2022). <https://doi.org/10.1613/jair.1.13676>
23. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) *CADE 27*. LNCS, vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
24. Schurr, H.: Optimal strategy schedules for everyone. In: Konev, B., Schon, C., Steen, A. (eds.) *Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022)*, Haifa, 11–12 August 2022. *CEUR Workshop Proceedings*, vol. 3201. CEUR-WS.org (2022). <https://ceur-ws.org/Vol-3201/paper8.pdf>
25. Seipp, J., Sievers, S., Helmert, M., Hutter, F.: Automatic configuration of sequential planning portfolios. In: Bonet, B., Koenig, S. (eds.) *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 25–30 January 2015, Austin, pp. 3364–3370. AAAI Press (2015). <https://doi.org/10.1609/AAAI.V29I1.9640>
26. Suda, M.: Vampire getting noisy: will random bits help conquer chaos? (system description). *EasyChair Preprint no. 7719* (2022). <https://easychair.org/publications/preprint/CSVF>
27. Sutcliffe, G.: The CADE ATP system competition - CASC. *AI Mag.* **37**(2), 99–101 (2016). <https://doi.org/10.1609/AIMAG.V37I2.2620>
28. Sutcliffe, G.: The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6.4.0. *J. Automat. Reason.* **59**(4), 483–502 (2017). <https://doi.org/10.1007/s10817-017-9407-7>
29. Suttner, C.B., Sutcliffe, G.: The CADE-14 ATP system competition. *J. Autom. Reason.* **21**(1), 99–134 (1998). <https://doi.org/10.1023/A:1006006930186>
30. Tammet, T.: Gandalf. *J. Autom. Reason.* **18**(2), 199–204 (1997). <https://doi.org/10.1023/A:1005887414560>
31. Tammet, T.: Gandalf c-1.1 (1998). <https://www.tptp.org/CASC/15/SystemDescriptions.html#Gandalf>. Accessed 25 Jan 2023
32. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) *CADE-15*. LNCS, vol. 1421, pp. 427–441. Springer, Cham (1998). <https://doi.org/10.1007/BFb0054276>

33. Urban, J.: BliStr: The blind strategymaker. In: Gottlob et al. [4], pp. 312–331. <https://doi.org/10.29007/8n7m>
34. Voronkov, A.: Spider: learning in the sea of options (2023). <https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833>. Unpublished. Paper accepted at Vampire23: The 7th Vampire Workshop
35. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, 18–20 May 1998, Sanibel Island, pp. 142–146. AAAI Press (1998). <http://www.aaai.org/Library/FLAIRS/1998/flairs98-027.php>
36. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, 11–15 July 2010. AAAI Press (2010). <http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1929>





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

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





Lemma Discovery and Strategies for Automated Induction

Sólrún Halla Einarsdóttir¹ , Márton Hajdu² , Moa Johansson¹ ,
Nicholas Smallbone¹ , and Martin Suda³ 

¹ Chalmers University of Technology, Gothenburg, Sweden

{slrn,jomoa,nicsma}@chalmers.se

² TU Wien, Vienna, Austria

marton.hajdu@tuwien.ac.at

³ Czech Technical University in Prague, Prague, Czech Republic

martin.suda@cvut.cz

Abstract. We investigate how the automated inductive proof capabilities of the first-order prover Vampire can be improved by adding lemmas conjectured by the QuickSpec theory exploration system and by training strategy schedules specialized for inductive proofs. We find that adding lemmas improves performance (measured in number of proofs found for benchmark problems) by 40% compared to Vampire’s plain structural induction as baseline. Strategy training alone increases the number of proofs found by 130%, and the two methods in combination provide an increase of 183%. By combining strategy training and lemma discovery we can prove more inductive benchmarks than previous state-of-the-art inductive proof systems (HipSpec and CVC4).

Keywords: Induction · Theory Exploration · Lemma Discovery · Strategies · Vampire

1 Introduction

We have experimented with augmenting Vampire’s capabilities for induction by injecting extra lemmas suggested by the theory exploration system QuickSpec [25] and by training strategy schedules specialized for inductive proofs. Our aim is to improve on the state of the art in automating proofs by induction.

Proofs by induction provide a challenge for automated theorem provers. Not only are there typically many choices of which induction scheme to use, but a proof may also require the conjecture to be generalized to strengthen the inductive hypothesis, or require additional auxiliary lemmas, themselves needing another induction to prove. For example, suppose we have a recursively defined function *rev* for reversing lists, defined using the append function *++*:

$$\begin{aligned} \text{rev } [] &= [] \\ \text{rev } (x : xs) &= (\text{rev } xs) ++ (x : []) \end{aligned}$$

where $++$ is defined as follows:

$$\begin{aligned} [] ++ xs &= xs \\ (x : xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

and want to prove that $rev(rev(xs)) = x$ for any list xs . When we ask Vampire to find a proof of this using structural induction it is unable to find a proof, even when given a long time. The induction hypothesis $rev(rev(xs)) = xs$ is not strong enough to prove that $rev(rev(x : xs)) = x : xs$: we are missing some *lemmas*.

QuickSpec [25] is a system that produces equational conjectures from function definitions. Suppose we use QuickSpec to conjecture some lemmas about the rev and $++$ functions. In under 1.5 s (running on a regular laptop¹) QuickSpec outputs the following 9 equations as unproved conjectures:

1. $rev [] = []$
2. $x ++ [] = x$
3. $[] ++ x = x$
4. $rev (rev x) = x$
5. $rev (x : []) = x : []$
6. $(x ++ y) ++ z = x ++ (y ++ z)$
7. $x : (y ++ z) = (x : y) ++ z$
8. $rev x ++ rev y = rev (y ++ x)$
9. $(xs ++ (y : (z : []))) = rev (z : (x : (rev xs)))$

Now suppose we add these equations to the input we give to Vampire, marking them as conjectured lemmas. Vampire may use such lemma in a proof, but only if it also proves it (e.g. by induction). Vampire instantly (in 6 ms, running on the same laptop) finds a proof of the original property, using (2), (6), and (8) above as lemmas, as well as proofs for the lemmas that were used. A closer investigation shows that only (6) and (8) are necessary to find a proof, where (8) is used in the proof of the original goal and (6) is used to prove (8).

Coming up with lemmas is a non-trivial task, and has sparked research into various lemma discovery techniques (see [17] for an overview). Lemma discovery can broadly be divided into two categories: *Top-down* techniques include attempting to generalize the current subgoal, or analyzing failed proof attempts to suggest a missing lemma. *Bottom-up* techniques focus on discovering potentially interesting lemmas about the definitions and concepts available, without considering any particular ongoing proof attempts. Bottom-up techniques can find a wider class of lemmas, but have the disadvantage that the system spends time working with conjectures that are not relevant to the goal. For example, the earlier system HipSpec [5] would first run QuickSpec (just as in the example above) but then attempt to prove *all* discovered conjectures before working on the main goal.

¹ The same laptop the experiments in Sect. 4 were run on, see more precise description there.

In this work, we use theory exploration, a bottom-up technique, in a more goal-directed manner. We use QuickSpec to suggest useful lemmas, but we will not prove *all* the suggestions, only those that are useful in the proof of the main goal. To do this we leverage Vampire’s AVATAR architecture [20,29], which allows us to attempt (speculatively, in parallel) the proof of the main goal using any subset of the candidate lemmas. Lemmas used must also be independently proved, but if that turns out to be hard (or even impossible) other options of finishing a proof may also be possible. Non-useful conjectures can be ignored and need not be proved, saving time. Since automatic theorem provers (ATPs) like Vampire and cvc5 now natively support applying automated induction [11,22] it is no longer necessary to use a specialized prover to apply induction before sending the resulting proof obligations to an ATP, as HipSpec did, and we examine the differences between the two approaches.

The performance of ATPs like Vampire is heavily influenced by the use of proving *strategies* and their combinations into *schedules* [15,27,28,30]. In addition to investigating the influence of adding lemmas from theory exploration, we also experiment with various learned strategies tailored for inductive proofs. A specialized strategy may allow Vampire to invent some easy lemmas itself, by applying generalization of a suitable subterm in a goal, lessening the need for theory exploration. However, finding strong targeted strategies is a time consuming endeavour which requires a set of problems with similar characteristics to those which we are interested in proving. For regular users, who typically just want to apply Vampire out of the box, this might not be an option.

2 Background

We propose the following design for an inductive theorem proving system:

1. We first use QuickSpec for theory exploration on the theory in question, generating equational conjectures about the theory.
2. The theory file including the original goal plus the conjectures from QuickSpec is sent to Vampire to attempt to find a proof.

Using our tools these two steps can be performed fully automatically, taking a problem file in the TIP [7] format as input and returning the proof found by Vampire as output.

2.1 QuickSpec

As seen in Sect. 1, QuickSpec is a system that produces equational *conjectures* about a theory. The conjectures are not guaranteed to be true, but have been tested to hold on 1000 randomly-generated test cases.² QuickSpec was originally designed to make conjectures about Haskell programs, but has been adapted to problems in inductive theorem proving.

² In automated reasoning terms, this means that 1000 ground instances of the conjecture have been shown to hold.

Conjecturing equations is difficult because of combinatorial explosion: even if we consider only quite simple equations and theories, there are many millions of possible conjectures. For example, if we identify a set of $n = 10,000$ interesting *terms*, then there are $n^2 = 100,000,000$ candidate equations which could be built from those terms. Generating and testing all of them is out of the question.

QuickSpec uses a more sophisticated approach which scales with the number of *terms* (e.g. 10,000) rather than the number of possible *equations* (e.g. 100,000,000). We enumerate terms in order of size (these terms may end up being the left or right hand side of an equational conjecture). We consider each term one by one, building up two sets as we go:

- The set of *discovered conjectures* between the terms considered so far.
- The set of *representative terms*. This consists of the set of terms considered so far, except that when several terms are equal, only one of them will be chosen as a representative. Therefore no two representative terms are equal.

Each time we consider a new term t , we answer the following question: *Is it equal to any representative term?* We do this in two steps:

1. *Pruning.* We check if the discovered conjectures imply that t is equal to a representative term r . If so, we simply ignore t and move on to the next term.
2. *Testing.* We test t against all representative terms. If it seems to be equal to some representative term r , we produce the conjecture $t = r$. Note that, since no two representative terms are equal, we only ever produce one conjecture per term.

If neither case holds, we add t as a representative term. The idea here is that, in case (1), the equation $t = r$ is redundant – we knew it already – whereas in case (2), it is new information and hence a potentially useful conjecture.

For example, suppose we take the list append function `++` and consider the following terms, where x, y and z range over lists: `[], x, y, z, x ++ [], y ++ [], x ++ (y ++ z), x ++ (x ++ y), (x ++ y) ++ z, (x ++ x) ++ y`.³ Initially, the set of discovered equations and the set of representative terms are empty. The algorithm proceeds as follows:

- `[]`. We add `[]` as a representative term.
- `x, y, z`. None of these terms are equal to each other or `[]`, so we add them as representative terms.
- `x ++ []`. The testing step reveals that this term is equal to x . We produce the conjecture (1): $x ++ [] = x$ (and do not add `x ++ []` as a representative term).
- `y ++ []`. The pruning step shows that this term is equal to y , by conjecture (1). We discard the term.
- `x ++ (y ++ z), x ++ (x ++ y)`. We add these terms as representatives.

³ In reality we enumerate terms in a systematic way, and a further refinement to the algorithm, *schemas*, eliminates many terms that differ from an existing term only in choice of variables.

- $(x ++ y) ++ z$. Testing shows that this term is equal to $x ++ (y ++ z)$. We produce the conjecture (2): $(x ++ y) ++ z = x ++ (y ++ z)$.
- $(x ++ x) ++ y$. Pruning shows that this term is equal to $x ++ (x ++ y)$, by conjecture (2). We discard the term.

At the end we have produced the conjectures (1) $x ++ [] = x$ and (2) $(x ++ y) ++ z = x ++ (y ++ z)$. Note that these conjectures are *complete* with respect to the enumerated terms, in the sense that any true equation between two such terms follows from the conjectures. In general, the QuickSpec algorithm produces a complete set of equations in this sense (though not necessarily sound, i.e. we may have false equations if we are unlucky in the testing).

It is perhaps not obvious why this algorithm should be fast. We point out the following reasons:

- The runtime of the algorithm scales with the *number of terms considered*, not the number of possible equations. This is because, with careful data structures and algorithms, in both the pruning and testing steps we can efficiently compare a new term against *all representative terms at once*, in close to constant time. For pruning, we use unifying completion [1] as implemented in Twee [24] to build a rewrite system from the discovered conjectures. We then keep the set of representative terms normalized with respect to this rewrite system. To prune a new term, we just normalize it and see if this normal form appears in the set of representative terms. For testing, we build a decision tree which allows us to, with a few (typically < 10 test cases), either show that a new term t is not equal to any representative term, or find precisely one term r that it *might* be equal to, whereupon we can test the single equation $t = r$ more thoroughly.
- In the common case, it takes a tiny amount of time ($\ll 1$ ms) to consider each term. That is because: (1) in the pruning step, the term is just normalized, an operation taking microseconds; (2) in the testing phase, typically the term is not equal to any representative term, in which case (as mentioned above) the term is evaluated on only a few test cases. The only expensive case is when testing reveals that the new term is equal to a representative case – but this is precisely the case where we have discovered a new conjecture!

Therefore, the runtime of QuickSpec largely grows proportionally with the *number of discovered conjectures*, plus a small amount which is proportional to the number of explored terms. In practice, QuickSpec is able to handle theories with ≈ 20 functions and generate equations having ≈ 10 symbols on each side, after which the number of discovered conjectures typically becomes too huge.

2.2 Induction in Vampire

Vampire supports induction over both term algebras and integers. The former, used in this work, is based on a constructor-style and two infinite descent-style schemas [21] in addition to ad hoc schemas generated from well-founded recursive

functions in the search space [13]. When inducting on a term in a unit clause (a literal), an instance of a schema with the negation of the unit clause is added to the search space. A stronger (and also more explosive) feature is non-unit induction, which inducts on arbitrarily many occurrences of a term, possibly across many literals and clauses.

Some basic lemma generation techniques such as generalizations over complex terms and occurrences [12] as well as active occurrence heuristics are also supported. In the presence of function definitions or induction hypotheses, (unordered) paramodulation may be used to reach lemmas otherwise not reachable with ordered superposition [13]. For a more detailed description of induction in Vampire we refer to [11].

Lemma Generation in Vampire. Vampire uses the traditional top-down backward reasoning approach to generate lemmas. It tries to reduce goals into subgoals and apply inferences on them, interleaved with induction inferences applied to all intermediate consequences that result from this process. A new lemma may be conjectured by generalizing over one of the terms in a subgoal. This lemma generation approach in Vampire usually derives different lemmas than QuickSpec's bottom-up theory exploration approach.

Simplifications and Orderings in Superposition. As superposition is tailored for first-order reasoning, it does not come as a surprise that some techniques that increase the efficiency of first-order reasoning are incompatible with inductive reasoning or higher-order reasoning in general. In particular, simplifications and orderings can affect a built-in induction within superposition.

Simplifications are inferences where one of the premises becomes redundant for further first-order reasoning and can be removed. For example, demodulation rewrites a clause into a smaller clause with an unconditional (unit) equation, and removes the original clause. In inductive reasoning things are not as simple, and any clause (even if it follows from smaller clauses) can be useful to generate interesting lemmas. For example, we might simplify a clause that would give rise to a crucial generalized lemma into a clause that does not give the same generalization anymore. Interestingly, given that simplification steps take up most of the inferences in a saturation run, in our experience this affects inductive reasoning less than expected.

3 Implementation

In order to perform our experiments we needed to integrate the lemmas conjectured by QuickSpec into Vampire's proof search, and choose a promising proof search strategy.

3.1 Conjectured Lemmas, AVATAR, and Vampire’s Claims

Integrating conjectured lemmas into proof search poses a technical challenge as they must be proven before they can be soundly used in a proof. At the same time, trying to prove each suggested lemma before the main goal is even attempted can create a great deal of unnecessary work. As has been noted before [8, 21], this challenge can be smoothly overcome in the presence of the AVATAR architecture for clause splitting [20, 29].

AVATAR keeps track of information about which clause has been derived from which splitting assumption, and soundly propagates it through inferences. Deriving the empty clause conditioned on some assumptions then does not necessarily mean the search is successfully concluded, but merely signifies that the conjunction of the attached assumptions can no longer be maintained. (AVATAR then updates its propositional model to reflect this newly derived information through a call to an underlying SAT or SMT solver.)

Let us assume we want to accommodate a speculative proof with lemmas L_1, \dots, L_n under AVATAR, where each lemma L_i is a closed formula. As a first approximation to explaining how this can be done, let us imagine introducing and immediately splitting the tautologies $L_i \vee \neg L_i$ for $i = 1, \dots, n$.⁴ Each clause in the search then carries (independently for each i) the information whether it depends on: 1) the assumption corresponding to L_i (proving with the help of lemma L_i), 2) $\neg L_i$ (trying to prove lemma L_i), or 3) neither of these (currently ignoring lemma L_i). Depending on the order in which (conditional) empty clauses get derived, the whole power set of possible scenarios is played out as if in parallel, in which some lemmas may already have been shown to suffice for proving the main conjecture, while themselves waiting to be proven (possibly with the help of other lemmas). The underlying SAT/SMT solver orchestrates the whole endeavour, decides which compatible subset of assumptions will be worked on next, and declares the proof attempt successful as soon as the first such scenario is complete. We remark that cyclic reasoning is automatically avoided by treating the assumptions of L_i and $\neg L_i$ as mutually exclusive.

It is surprisingly easy to get access to this feature of speculative lemma use in Vampire under AVATAR. In fact, we can rely on a small adjustment of just the parser added by Andrei Voronkov already in 2011. In the TPTP language [26], this adjustment introduced a new custom formula role called the *claim*. Precisely as in our use case, a claim is a formula that most likely follows from the surrounding axioms and has a high chance of being useful for proving the given conjecture, but must be itself also proven by the system in a valid proof. For this work we extended Vampire’s SMT-LIB parser in an analogous way and added a custom construct `assert-claim` with the same semantics.

⁴ In reality, both L_i and $\neg L_i$ must also be skolemized and clasified, which in the prover happens before splitting. We return to this aspect further below.

Technically, when the parser reads a claim formula L , it picks a fresh propositional symbol p_L and passes on the equivalence $p_L \leftrightarrow L$ as a standard axiom.⁵ The equivalence $p_L \leftrightarrow L$ is then classified to

$$\{\neg p_L \vee C \mid C \in \text{CNF}(L)\} \quad \text{and} \quad \{p_L \vee D \mid D \in \text{CNF}(\neg L)\}.$$

AVATAR recognizes the p_L and $\neg p_L$ as complementary ground components and will then always assert either p_L or $\neg p_L$. Thus the first-order part of the prover must work with either the clauses from $\text{CNF}(L)$ or from $\text{CNF}(\neg L)$, while AVATAR keeps track of the respective dependencies.

3.2 Proving Strategies and a New Induction Schedule

A theorem prover typically has many parameters (in Vampire called *options*) that can be changed to adjust the proof search characteristics. In Vampire, there are more than 100 options for configuring the preprocessing steps, the saturation algorithm, generating and simplification rules, proof search heuristics and also induction. By a *strategy* we mean a concrete assignment of values to such options. It is long known [27, 31] that the success rate of an ATP can be dramatically improved by arranging a number of different proving strategies of complementary characteristics into a *strategy schedule*, a sequence of strategies with assigned time budgets, to be executed in sequence (or in parallel).

In this work, we constructed a strategy schedule specifically targeting inductive theorem proving on the TIP benchmarks (see Sect. 4.1). We followed the strategy discovery recipe pioneered by the Spider system [30]. This consists of

1. Randomly sampling strategies to try to solve a previously unsolved problem (or possibly to improve the solution time on a problem already known to be solvable).
2. Optimizing the found strategy on that problem using local search (in which, for each option in turn, different values are tried out and a new value is committed to, if the corresponding change leads to an improved time or the time stays the same, but the value becomes default).
3. Evaluating the optimized strategy on all problems, to update the information about which problems are solvable and in what best time.

In our case, we sampled strategies from a space defined by a total of 115 base Vampire options and 19 dedicated induction options. Most of these options are Boolean, many are finite enumerations of discrete values and a few are numeric. It is clear that the totality of all strategies is astronomically large and random sampling is a way to have access to all the strategies, at least in principle. We searched for strategies in parallel on 60 cores of our server⁶ for several days. In

⁵ The only extra effort is to mark and protect the new symbol p_L against potential elimination during preprocessing, as, after all, the new equivalence would otherwise qualify as an unused predicate definition and could be discarded.

⁶ Equipped With Intel®Xeon®Gold 6140 CPU @ 2.3 GHz and 500 GB RAM.

the end, we collected 246 strategies covering 236 of the 486 TIP benchmarks that we used for training.

Once a sufficiently large set of strategies has been discovered (or when the rate of solving new problems becomes too low to make search for additional strategies worth the effort), schedule construction can be formulated as an integer programming task, in which running times are assigned to individual strategies to cover the union of as many problems as possible while not exceeding a given overall time bound [15, 23]. We instead adopted a greedy algorithm [4] to a weighted set cover formulation of the problem: starting from an empty schedule, we iteratively add a new strategy s for additional t units of time if this step is currently the best in terms of $1/t$ —“the number of problems that will additionally get covered”. This greedy approach does not guarantee an optimal result, but runs in polynomial time and is really easy to implement. (See also [3].)

Our final schedule makes use of 66 of the discovered strategies and should be able to solve all of the covered 236 problems in under 12s (per problem). For our later experiment we prepared a second schedule, specialized to also take into consideration the versions of the TIP benchmarks with the added lemmas (cf. label T in Sect. 4 below). This schedule makes use of 86 strategies, aims to cover a total of 522 problems (version with and without lemmas counted separately) and runs to completion after approximately 24s.

4 Evaluation

In our evaluation we compare several variants of Vampire. We start with two baseline versions without strategy scheduling:

- (V): Vampire with the following flags for structural induction:

```
-ind struct -indoct on -nui on -to lpo -drc off
```

The option `-ind struct` enables using structural induction (constructor-based induction axioms for term algebras), and with `-indoct on` these axioms are based on generalizing over any term, not just Skolem constants. Moreover, the induction axioms are generated from any clause set using `-nui on`. Finally, `-to lpo` and `-drc off` enable a simplification ordering which is well-suited for handling recursive functions.

- (V + L): Vampire with the same flags active as in (V), plus conjectures from QuickSpec added to the problem files as claims as described in Sect. 3.1.

The idea is that (V) serves as a baseline for what kinds of inductive proofs Vampire is capable of. By comparing (V) with (V + L), we see whether the lemmas discovered by QuickSpec help Vampire.

Next we add versions of Vampire with specialized strategy schedules:

- (S): Vampire with the specialized strategy schedule for inductive problems described in Sect. 3.2.

- (S + L): Vampire with both the strategy schedule as in (S) above and conjectures from QuickSpec added to the problem files as claims.

By comparing (S) with (V+L), we can see the relative importance of strategy scheduling and lemmas. In (S+L) we can see whether the two strategies complement each other. We expect that (S) may see less benefit from lemmas than (V) because the learned strategies may be better at e.g. generalising subgoals. Note that the strategy schedule is tuned without seeing the lemmas, so it is even possible that (S+L) might perform worse than (S) due to the extra lemmas disturbing the strategy scheduler.

Since the strategy schedule is tuned without seeing the lemmas, (S+L) illustrates what we can get by taking an existing prover with a built-in strategy schedule, and adding new lemmas to it. Notice also that any problems that require lemmas will not be proved during training, so will not influence the schedule. We can use these problems as a kind of test set for S+L, as they are effectively unseen during training.

To investigate the limits of our approach we add a third family:

- (T): Vampire with a specialized strategy schedule for inductive problems, trained on the TIP problems after conjectures from QuickSpec have been injected into the problem files.
- (T + L): Vampire with the lemma-specialized schedule (T) as above and conjectures from QuickSpec added to the problem files as claims.

Note that the strategy used by (T) and (T+L) may be prone to overfitting, as all the test problems are seen during training and influence the schedule.⁷ The results for (T) and (T+L) are useful as a benchmark to compare the other provers against, and an indication of what a perfectly-tuned strategy schedule could do.

We evaluated our methods on the TIP benchmark set. For all methods the time limit was set to 30s. Since the strategy schedules are randomized and may not find the same proofs every time they are run, we ran each one 5 times on each problem. The experiments were run on a Dell Inc. Latitude 5320 with an 11th Gen Intel®Core™i5-1145G7 @ 2.60GHz × 8 processor and 16GB RAM. Scripts used to run experiments and process results are available at <https://github.com/solrun/vampspec>.

4.1 TIP Benchmarks

TIP is a collection of benchmarks specifically for inductive theorem provers [6]. The problems are expressed in a syntax very similar to SMT-LIB [2], and come

⁷ Why not use a training/test split? Because there are not very many problems in total, and more importantly, because many problems are related, which makes it hard to design an uncontaminated test set, since we need to avoid having related problems where one is in the training set and one is in the test set.

with tools to translate the problems into various formats (including standard SMT-LIB) as well as built-in support for lemma generation using QuickSpec.

TIP consists of several subsets: the *prod* set contains 50 theorems and 24 lemmas about lists and natural numbers defined in [16], the *IsaPlanner* set defined in [18] contains 86 properties originally designed to test provers that use the rippling heuristic. The *prod* and *IsaPlanner* problems have previously been used to evaluate a number inductive theorem provers [5, 9, 22] so experiments with them enable comparison to previous work.

The *TIP2015* set contains a further 326 problems and was added as many existing provers, like HipSpec, could solve almost all problems in the previous two sets. It includes a variety of problems such as various sorting algorithms with correctness properties expressed in alternative ways, properties of regular expressions, binary search trees, integers implemented on top of natural numbers, natural numbers in binary representation, and properties of various functions on lists and natural numbers. Some of the problems were not known to have been automated at the time of their publication [6] and are offered as challenges.

4.2 Results

Table 1 shows the number of proofs found for the 486 TIP benchmarks, by the different methods that we described previously. We count a proof as found if it was found in any of the 5 proof attempts using that strategy. We found that Vampire with structural induction enabled, (V), finds proofs to 102 of the problems, which increases to 143 with the addition of lemmas from QuickSpec. The specialized strategy schedule, (S), finds 236 proofs, more than twice as many as (V). The specialized strategy schedule finds some more proofs with the addition of lemmas but the increase is not so great, from 236 to 263. The strategy schedule trained on problems already containing lemmas, (T), finds 237 proofs (the same proofs as (S) and one additional proof), which increases to 288 with the addition of lemmas. The bottom line of Table 1 shows the number of proofs found by each method with or without lemmas added.

Table 1. The number of proofs found for the 486 TIP benchmarks when testing the different proof methods in the presence and absence of generated lemmas.

Proofs found	(V)	(S)	(T)
no lemmas	102	236	237
with lemmas (+ L)	143	263	288
Total proofs found	153	269	289

Although all methods find more proofs with lemmas than without, a number of proofs can only be found without the additional lemmas and are lost after lemmas are added. Most often when using ATPs, different strategies or parameterizations might both gain and lose some proofs rather than one simply

being strictly better than the other. Table 2 shows this for our three strategies, with and without added lemmas. As mentioned above there are a small number of proofs (10 for (V), 6 for (S) and 1 for (T)) which are found by the each strategy without lemmas, but not after lemmas are added. Since the added lemmas increase the size of the proof search space, we are not surprised that they may in some cases prevent the strategy from finding a proof in time. In the case of the specialized strategy schedule (T) which has already seen the problems with added lemmas in its training, the added lemmas only hinder it from finding a proof in one instance. In the case where (T+L) loses the proof (T) found, *TIP2015/regexp_RecAtom*, the number of conjectured lemmas added to the problem file is very large (459) which probably causes the search space to explode. For this particular problem both strategies (V) and (S) also found a proof, but no strategy found a proof for the problem with lemmas added.

Note that (T+L) finds 53 proofs not found by (S), showing the improvement achievable by adding QuickSpec’s lemma conjecturing and using a strategy schedule specialized to make use of those lemmas, compared to only using strategy schedule training as with (S). Of these 53 problems, (S+L) finds proofs for 33 of them, making use of the added lemmas without having seen them in its training. As mentioned above, the strategy schedule of (S+L) is effectively not trained on any problems that require lemmas, so we can view the 53 problems as test problems, unseen in the training data, all solvable with a perfect strategy, and say that (S+L) solves 62% of those problems. Thus we get an indication that the strategy schedule is generalizing to unseen problems.

Table 2. Here each column shows the number of unique proofs found by the respective method (column label) but not by one of the other methods (row label).

	(V)	(V + L)	(S)	(S + L)	(T)	(T + L)
- (V)		51	134	162	135	187
- (V + L)	10		109	124	109	145
- (S)	0	16		33	1	53
- (S + L)	1	4	6		6	25
- (T)	0	15	0	32		52
- (T + L)	1	0	1	0	1	

In some cases, one of our methods performs strictly better than another, namely (S) and (T) are strictly better than (V), (T) is strictly better than (S), and (T+L) is strictly better than (S+L). Since (S) and (T) are specialized strategy schedules that can execute many different strategies for each proof attempt, including the strategy used by (V), it is unsurprising that they subsume (V). Since (S) and (T) are strategy schedules trained in the same manner, with the only difference being that (T) is trained on a superset of the problems (S) is trained on, and both evaluated on problems they have encountered in

their training, we expect them to achieve a similar performance. Note that (T) only finds one proof not found by (S), so their performance is nearly equivalent. Since (T+L) is evaluated on problems with added lemmas that it has already seen during training while (S+L) is given previously unseen lemmas in its input problems, it would be surprising if (S+L) found a proof that (T+L) could not. In all cases these pairs of methods are evaluated on the same input problems (both are evaluated on problems with additional lemmas or both on the problems without lemmas).

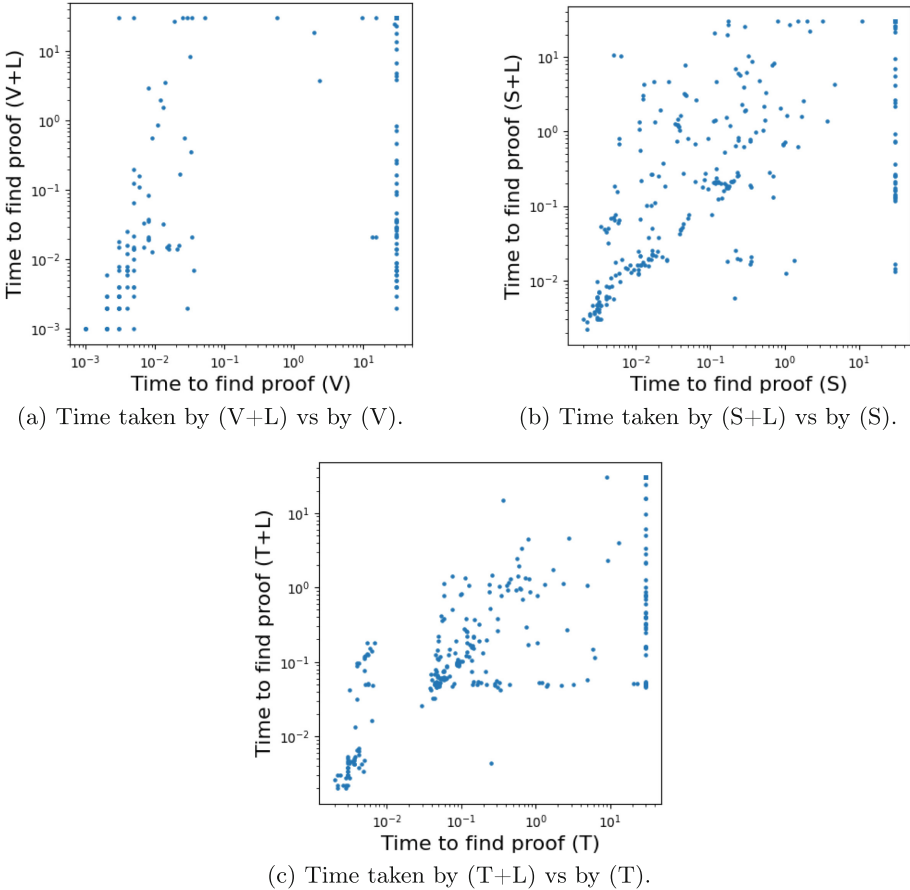


Fig. 1. Time taken to find a proof with lemmas versus without them using the same strategy (on a log scale).

In cases where the same strategy could find a proof both with and without lemmas added to the problem file, we compare the time taken to find a proof as a metric of how easy the proof is to find. Figure 1 shows the plots of the time taken to find a proof with lemmas versus without them using the same strategy (on a

log scale). The points around the edges indicate that the respective method did not find a proof within the given time limit (30s), so points along the right-hand edge indicate that a proof was found with lemmas and not without them, while points along the top edge indicate a proof was found without lemmas and lost after they were added.

For problems where both (V) and (V+L) found a proof, the average time for (V) was 0.67s with a standard deviation of 3.56s, while the average time for (V+L) was 1.04s with a standard deviation of 4.30s. We can see how in most cases where a proof was found both with and without lemmas added, the proof search went faster without them, indicated by how most of the points are to the left of the diagonal. For problems where both (S) and (S+L) found a proof, the average time for (S) was 0.20s with a standard deviation of 0.49s while the average time for (S+L) was 1.53s with a standard deviation of 4.16s. We see many points clustered around the diagonal, indicating both proof searches took a similar amount of time to find a proof, though many more points lie to the left of the diagonal than to its right, indicating a faster proof search without lemmas. For problems where both (T) and (T+L) found a proof, the average time for (T) was 0.59s with a standard deviation of 2.39s while the average time for (T+L) was 0.37s with a standard deviation of 1.16s, so as opposed to methods (V) and (S), the proof time goes down with the addition of lemmas. Since the schedule here was trained on problems containing lemmas, it prioritizes strategies that make use of the available lemmas, thus finding the proofs more efficiently with lemmas.

Table 3. The number of proofs found in different subsets of the TIP benchmarks, along with results for CVC4 and HipSpec for the same subsets.

Set (size)	(V)	(V+L)	(S)	(S+L)	(T)	(T+L)	CVC4	HipSpec
<i>prod</i> (50)	7	29	27	46	28	49	39	47
<i>IsaPlanner</i> (85)	41	43	73	75	73	81	80	80
<i>TIP2015</i> (326)	39	53	113	119	113	134	–	–

The problems from the *IsaPlanner* and *prod* subsets of TIP were also used for evaluation of HipSpec in [5]⁸ and of inductive reasoning with CVC4 in [22]. The number of proofs they found are included in Table 3 along with the results of our experiments for those subsets.⁹ We see a clear difference in results on the *prod*-

⁸ In order to investigate whether the numbers for HipSpec would be better on a modern machine, we re-ran it on the *prod* benchmark. We found that it solved *fewer* problems, 44 in all, as a result of slight changes in HipSpec since the publication of [5]. No problems were solved by HipSpec today that were not solved back then.

⁹ We tried but failed to run HipSpec on the *TIP2015* problems. HipSpec’s input format is a limited dialect of Haskell and, while TIP problems can be converted to Haskell, the dialect is not the same as HipSpec’s. As HipSpec is unmaintained, we were unable to go further.

subset, which is designed such that more complicated lemmas are needed for most proofs, and the *IsaPlanner*-subset, which contains easier problems which can often be solved without external lemmas (or with just lemmas coming from generalizations of a subgoal). On both subsets we only achieve results competitive with either CVC4 or HipSpec when we combine a specialized strategy schedule with lemmas (S+L) and (T+L). On the *TIP2015* subset, none of the methods we tested found proofs for even half of the problems, and we leave a closer examination of what is required to achieve better results there as future work.

As described in Sect. 3.2, the strategy schedules may not find the same proofs in every run. In our experiments we ran each schedule 5 times and found there was a handful of problems where the same strategy schedule would sometimes find a proof and not others, the exact numbers are shown in Table 4. In the results shown in Tables 1–3 we count that a proof was found if it was found in at least one of the five runs.

Table 4. Number of inconsistently found proofs by each strategy schedule with and without added lemmas.

Method	(S)	(S + L)	(T)	(T + L)
Inconsistent Proofs	1	9	4	6

5 Discussion

Modern day ATPs like Vampire have many moving parts. Slight changes in configuration often lead to some extra proofs being found while others are lost. This is particularly true when considering also proofs by induction, as here the potential for exploding the search space in unproductive directions is even larger. It is often difficult to know in advance what parameters and strategies will affect the capabilities of finding a proof within reasonable time.

Our first experiments tested the effect of adding lemma candidates for inductive proofs to a standard out-of-the-box variant of Vampire, simulating what a regular user might have at hand. Here, we see a clear improvement in the number of proofs for the *TIP prod*-subset, where more complicated lemmas are needed for most of the proofs, and a modest improvement on other subsets. Still, the results are well below both CVC4 and HipSpec. We conclude that simply adding lemmas from QuickSpec to Vampire (with a default induction strategy) is not sufficient to reach a state-of-the-art performance.

Secondly, we also experimented with training specialized strategies, customized to inductive proofs on *TIP* problems. This seems to be necessary for top performance. We experiment with two trained strategies: the first on proofs of *TIP*-problems without added lemmas (starting from the out-of-the-box Vampire setup). The second, to get an upper bound of how well Vampire could perform, we also trained on proofs *with the added lemmas* from QuickSpec. With a customized strategy for induction, already without lemmas Vampire performs much

better than before. We notice that the increase is larger for the customized strategy than it was for adding lemmas to the standard Vampire version! We conclude that specialized strategies have a larger effect on the number of proofs than just adding auxiliary lemmas.

Finally, we added the QuickSpec lemmas. Both strategies now improved even more, especially on the TIP *prod*-subset, where they both beat previous state of the art, proving 46 and 49 problems respectively. As expected, the strategy trained on proofs with lemmas (T) had a larger increase, being able to use the lemmas available more efficiently. Interestingly, on the TIP *IsaPlanner*-subset, only the (T) strategy beat the state of the art. We conclude that in the presence of auxiliary lemmas and together with specialized strategies, Vampire can indeed outperform previous state of the art systems CVC4 and HipSpec.

However, one might argue that the comparison is biased. To get state-of-the-art performance from Vampire requires a strategy optimised by seeing and trying the problems already! This might not be a viable option for all users. We do not know how well these schedules would perform on other types of inductive problems as they are likely overfitted to TIP to some degree. We could have divided the TIP problems into training and testing sets to try to avoid overfitting, but the TIP set is not very large, only 486 problems (of which only 60% could be solved using any method we tried), and many problems are similar to each other, so it is not clear that this would solve the problem. In short, there is too little data to train a general-purpose strategy, and we can not say how well the learned strategy generalizes to problems outside of TIP.

Even so, domain-specific strategies are reasonable in many applications. For example, in program verification, it is reasonable to run the prover over a set of problems multiple times, and find a strategy that works for just those kind of problems one is interested in verifying. Our results show that specially-tuned strategies are highly effective, and compatible with lemma discovery.

The search space for proofs where induction is allowed is inherently enormous and becomes particularly explosive when the ATP itself has to decide when to apply induction. Trained strategies seem to be necessary for competitive performance. HipSpec on the other hand was developed before CVC4 and Vampire supported induction, and thus handled the induction step outside the ATP, and only outsourced the resulting subgoals. One benefit of doing so is that the search space is much less explosive, which contributes to HipSpec's good performance. We thus leave the question of how to best implement automated induction partially unsolved: we either need highly specialized strategies trained on many attempts of proofs, or keeping the application of induction under strict control.

5.1 Future Work

We have many ideas for improvements when it comes to generating lemmas for inductive proofs. QuickSpec is limited to discovering equational conjectures that may have a predicate as a condition (if the theory being explored contains a function that returns a boolean value, the value of that function may be used as a predicate). However, many inductive proofs require more complex conditional

lemmas. In [10] we presented RoughSpec, a system that generates conjectures that match a user-defined input template. This could be used to conjecture lemmas for inductive proofs, using lemma templates likely to be useful learned from proof libraries. Another idea is developing better methods of only providing lemmas likely to be useful, limiting the number of lemmas given to the prover so that the search space does not explode. For example, there are some prominent examples of using simple syntactic conditions [14] inside the theorem prover or using machine learning [19] before the invocation of the theorem prover to mitigate this issue.

Acknowledgments. This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation. Martin Suda was supported by the Czech Science Foundation project no. 24-12759S and the project RICAIP no. 857306 under the EU-H2020 programme.

References

1. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: *Rewriting Techniques*, pp. 1–30. Elsevier (1989)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
3. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction. In: *IJCAR (2024)*, accepted
4. Chvátal, V.: A greedy heuristic for the set-covering problem. *Math. Oper. Res.* 4(3), 233–235 (1979)
5. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) *CADE 2013*. LNCS (LNAI), vol. 7898, pp. 392–406. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_27
6. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) *CICM 2015*. LNCS (LNAI), vol. 9150, pp. 333–337. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_23
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: The TIP format. <http://tip-org.github.io/format.html>
8. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) *FroCoS 2017*. LNCS (LNAI), vol. 10483, pp. 172–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_10
9. Dixon, L., Johansson, M.: *Isaplaner 2: A proof planner for isabelle* (2007)
10. Einarsdóttir, S.H., Smallbone, N., Johansson, M.: Template-based theory exploration: Discovering properties of functional programs by testing. In: *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages, IFL 2020*, pp. 67–78. Association for Computing Machinery, New York (2021). <https://doi.org/10.1145/3462172.3462192>
11. Hajdu, M., Hozzová, P., Kovács, L., Reger, G., Voronkov, A.: Getting Saturated with Induction, pp. 306–322. Springer Nature Switzerland, Cham (2022)

12. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Benzmüller, C., Miller, B. (eds.) *Intelligent Computer Mathematics*, pp. 123–137. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_8
13. Hajdú, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: *Formal Methods in Computer Aided Design, FMCAD 2021*, New Haven, CT, USA, 19–22 October 2021, pp. 1–10. IEEE (2021)
14. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) *CADE 2011. LNCS (LNAI)*, vol. 6803, pp. 299–314. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_23
15. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) *CICM 2021. LNCS (LNAI)*, vol. 12833, pp. 107–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81097-9_8
16. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. *J. Autom. Reason.* **16**, 79–111 (1996)
17. Johansson, M.: Lemma discovery for induction. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) *CICM 2019. LNCS (LNAI)*, vol. 11617, pp. 125–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23250-4_9
18. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: *International Conference on Interactive Theorem Proving* (2010)
19. Kühlwein, D., Blanchette, J.C., Kaliszyk, C., Urban, J.: MaSh: machine learning for sledgehammer. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) *Interactive Theorem Proving*, pp. 35–50. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_6
20. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: Felty, A.P., Middeldorp, A. (eds.) *CADE 2015. LNCS (LNAI)*, vol. 9195, pp. 399–415. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_28
21. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: *CADE* (2019). <https://api.semanticscholar.org/CorpusID:126940163>
22. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D’Souza, D., Lal, A., Larsen, K.G. (eds.) *VMCAI 2015. LNCS*, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5
23. Schurr, H.: Optimal strategy schedules for everyone. In: Konev, B., Schon, C., Steen, A. (eds.) *Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022)*, Haifa, Israel, 11 - 12 August, 2022. *CEUR Workshop Proceedings*, vol. 3201. CEUR-WS.org (2022). <https://ceur-ws.org/Vol-3201/paper8.pdf>
24. Smallbone, N.: Twee: An equational theorem prover. In: *CADE*, pp. 602–613 (2021)
25. Smallbone, N., Johansson, M., Claessen, K., Alghed, M.: Quick specifications for the busy programmer. *J. Funct. Program.* **27** (2017)
26. Sutcliffe, G.: The Logic Languages of the TPTP World. *Logic J. IGPL* (2022). <https://doi.org/10.1093/jigpal/jzac068>
27. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) *CADE 1998. LNCS*, vol. 1421, pp. 427–441. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054276>
28. Urban, J.: Blistr: The blind strategymaker. In: Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.) *Global Conference on Artificial Intelligence, GCAI 2015*, Tbilisi, Georgia, 16–19 October 2015. *EPiC Series in Computing*, vol. 36, pp. 312–319. EasyChair (2015), https://easychair.org/publications/volume/GCAI_2015

29. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46
30. Voronkov, A.: Spider: learning in the sea of options. In: Vampire23: The 7th Vampire Workshop (2023), <https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833>, to appear
31. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, May 18-20, 1998, Sanibel Island, Florida, USA, pp. 142–146. AAAI Press (1998). <http://www.aaai.org/Library/FLAIRS/1998/flairs98-027.php>

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

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





Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper)

Nils Lommen^(✉) , Éléonore Meyer , and Jürgen Giesl 

RWTH Aachen University, Aachen, Germany
{lommen, eleanore.meyer, giesl}@cs.rwth-aachen.de

Abstract. Recently, we showed how to use control-flow refinement (CFR) to improve automatic complexity analysis of integer programs. While up to now CFR was limited to classical programs, in this paper we extend CFR to *probabilistic* programs and show its soundness for complexity analysis. To demonstrate its benefits, we implemented our new CFR technique in our complexity analysis tool KoAT.

1 Introduction

There exist numerous tools for complexity analysis of (non-probabilistic) programs, e.g., [2–6, 10, 11, 15, 16, 18, 19, 24, 25, 28, 30, 32]. Our tool KoAT infers upper runtime and size bounds for (non-probabilistic) integer programs in a modular way by analyzing subprograms separately and lifting the obtained results to global bounds on the whole program [10]. Recently, we developed several improvements of KoAT [18, 24, 25] and showed that incorporating control-flow refinement (CFR) [13, 14] increases the power of automated complexity analysis significantly [18].

There are also several approaches for complexity analysis of *probabilistic* programs, e.g., [1, 7, 9, 21–23, 27, 29, 31, 34]. In particular, we also adapted KoAT’s approach for runtime and size bounds, and introduced a modular framework for automated complexity analysis of probabilistic integer programs in [27]. However, the improvements of KoAT from [18, 24, 25] had not yet been adapted to the probabilistic setting. In particular, we are not aware of any existing technique to combine CFR with complexity analysis of probabilistic programs.

Thus, in this paper, we develop a novel CFR technique for probabilistic programs which could be used as a black box by every complexity analysis tool. Moreover, to reduce the overhead by CFR, we integrated CFR natively into KoAT by calling it on-demand in a modular way. Our experiments show that CFR increases the power of KoAT for complexity analysis of probabilistic programs substantially.

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL.

© The Author(s) 2024

C. Benz Müller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 233–243, 2024.

https://doi.org/10.1007/978-3-031-63498-7_14

The idea of CFR is to gain information on the values of program variables and to sort out infeasible program paths. For example, consider the probabilistic **while**-loop (1). Here, we flip a (fair) coin and either set x to 0 or do nothing.

$$\mathbf{while} \ x > 0 \ \mathbf{do} \ x \leftarrow 0 \oplus_{1/2} \ \mathbf{noop} \ \mathbf{end} \quad (1)$$

The update $x \leftarrow 0$ is in a loop. However, after setting x to 0, the loop cannot be executed again. To simplify its analysis, CFR “unrolls” the loop resulting in (2).

$$\begin{aligned} &\mathbf{while} \ x > 0 \ \mathbf{do} \ \mathbf{break} \ \oplus_{1/2} \ \mathbf{noop} \ \mathbf{end} \\ &\mathbf{if} \ x > 0 \ \mathbf{then} \ x \leftarrow 0 \ \mathbf{end} \end{aligned} \quad (2)$$

Here, x is updated in a separate, *non-probabilistic* **if**-statement and the loop does not change variables. Thus, we sorted out paths where $x \leftarrow 0$ was executed repeatedly. Now, techniques for probabilistic programs can be used for the **while**-loop. The rest of the program can be analyzed by techniques for non-probabilistic programs. In particular, this is important if (1) is part of a larger program.

We present necessary preliminaries in Sect. 2. In Sect. 3, we introduce our new control-flow refinement technique and show how to combine it with automated complexity analysis of probabilistic programs. We conclude in Sect. 4 by an experimental evaluation with our tool KoAT. We refer to [26] for further details on probabilistic programs and the soundness proof of our CFR technique.

2 Preliminaries

Let \mathcal{V} be a set of variables. An *atom* is an inequation $p_1 < p_2$ for polynomials $p_1, p_2 \in \mathbb{Z}[\mathcal{V}]$, and the set of all atoms is denoted by $\mathcal{A}(\mathcal{V})$. A *constraint* is a (possibly empty) conjunction of atoms, and $\mathcal{C}(\mathcal{V})$ denotes the set of all constraints. In addition to “<”, we also use “≥”, “=”, etc., which can be simulated by constraints (e.g., $p_1 \geq p_2$ is equivalent to $p_2 < p_1 + 1$ for integers).

For *probabilistic integer programs (PIPs)*, as in [27] we use a formalism based on transitions, which also allows us to represent **while**-programs like (1) easily. A PIP is a tuple $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$ with a finite set of program variables $\mathcal{PV} \subseteq \mathcal{V}$, a finite set of locations \mathcal{L} , a fixed initial location $\ell_0 \in \mathcal{L}$, and a finite set of general transitions \mathcal{GT} . A *general transition* $g \in \mathcal{GT}$ is a finite set of transitions which share the same start location ℓ_g and the same guard φ_g . A *transition* is a 5-tuple $(\ell, \varphi, p, \eta, \ell')$ with a *start location* $\ell \in \mathcal{L}$, *target location* $\ell' \in \mathcal{L} \setminus \{\ell_0\}$, *guard* $\varphi \in \mathcal{C}(\mathcal{V})$, *probability* $p \in [0, 1]$, and *update* $\eta : \mathcal{PV} \rightarrow \mathbb{Z}[\mathcal{V}]$. The probabilities of all transitions in a general transition add up to 1. We always require that general transitions are pairwise disjoint and let $\mathcal{T} = \bigsqcup_{g \in \mathcal{GT}} g$ denote the set of all transitions. PIPs may have *non-deterministic branching*, i.e., the guards of several transitions can be satisfied. Moreover, we also allow *non-deterministic (temporary) variables* $\mathcal{V} \setminus \mathcal{PV}$. To simplify the presentation, we do not consider transitions with individual costs and updates which use probability distributions, but the approach can easily be extended accordingly. From now on, we fix a PIP $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$.

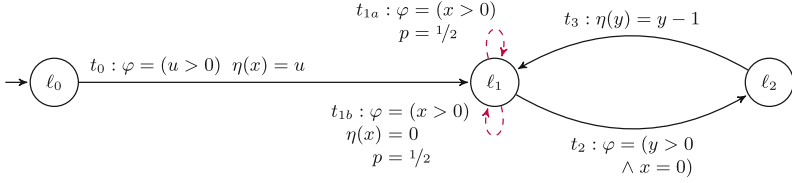


Fig. 1. A Probabilistic Integer Program

Example 1. The PIP in Fig. 1 has $\mathcal{PV} = \{x, y\}$, $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, and four general transitions $\{t_0\}$, $\{t_{1a}, t_{1b}\}$, $\{t_2\}$, $\{t_3\}$. The transition t_0 starts at the initial location ℓ_0 and sets x to a non-deterministic positive value $u \in \mathcal{V} \setminus \mathcal{PV}$, while y is unchanged. (In Fig. 1, we omitted unchanged updates like $\eta(y) = y$, the guard **true**, and the probability $p = 1$ to ease readability.) If the general transition is a singleton, we often use transitions and general transitions interchangeably. Here, only t_{1a} and t_{1b} form a non-singleton general transition which corresponds to the program (1). We denoted such (probabilistic) transitions by dashed arrows in Fig. 1. We extended (1) by a loop of t_2 and t_3 which is only executed if $y > 0 \wedge x = 0$ (due to t_2 's guard) and decreases y by 1 in each iteration (via t_3 's update).

A *state* is a function $\sigma : \mathcal{V} \rightarrow \mathbb{Z}$, Σ denotes the set of all states, and a *configuration* is a pair of a location and a state. To extend finite sequences of configurations to infinite ones, we introduce a special location ℓ_\perp (indicating termination) and a special transition t_\perp (and its general transition $g_\perp = \{t_\perp\}$) to reach the configurations of a run after termination. Let $\mathcal{L}_\perp = \mathcal{L} \uplus \{\ell_\perp\}$, $\mathcal{T}_\perp = \mathcal{T} \uplus \{t_\perp\}$, $\mathcal{GT}_\perp = \mathcal{GT} \uplus \{g_\perp\}$, and let $\text{Conf} = (\mathcal{L}_\perp \times \Sigma)$ denote the set of all configurations. A *path* has the form $c_0 \rightarrow_{t_1} \dots \rightarrow_{t_n} c_n$ for $c_0, \dots, c_n \in \text{Conf}$ and $t_1, \dots, t_n \in \mathcal{T}_\perp$ for an $n \in \mathbb{N}$, and a *run* is an infinite path $c_0 \rightarrow_{t_1} c_1 \rightarrow_{t_2} \dots$. Let Path and Run denote the sets of all paths and all runs, respectively.

We use Markovian schedulers $\mathfrak{S} : \text{Conf} \rightarrow \mathcal{GT}_\perp \times \Sigma$ to resolve all non-determinism. For $c = (\ell, \sigma) \in \text{Conf}$, a *scheduler* \mathfrak{S} yields a pair $\mathfrak{S}(c) = (g, \tilde{\sigma})$ where g is the next general transition to be taken (with $\ell = \ell_g$) and $\tilde{\sigma}$ chooses values for the temporary variables such that $\tilde{\sigma} \models \varphi_g$ and $\sigma(v) = \tilde{\sigma}(v)$ for all $v \in \mathcal{PV}$. If \mathcal{GT} contains no such g , we obtain $\mathfrak{S}(c) = (g_\perp, \sigma)$. For the formal definition of Markovian schedulers, we refer to [26].

For every \mathfrak{S} and $\sigma_0 \in \Sigma$, we define a probability mass function $pr_{\mathfrak{S}, \sigma_0}$. For all $c \in \text{Conf}$, $pr_{\mathfrak{S}, \sigma_0}(c)$ is the probability that a run with scheduler \mathfrak{S} and the initial state σ_0 starts in c . So $pr_{\mathfrak{S}, \sigma_0}(c) = 1$ if $c = (\ell_0, \sigma_0)$ and $pr_{\mathfrak{S}, \sigma_0}(c) = 0$ otherwise.

For all $c, c' \in \text{Conf}$ and $t \in \mathcal{T}_\perp$, let $pr_{\mathfrak{S}}(c \rightarrow_t c')$ be the probability that one goes from c to c' via the transition t when using the scheduler \mathfrak{S} (see [26] for the formal definition of $pr_{\mathfrak{S}}$). Then for any path $f = (c_0 \rightarrow_{t_1} \dots \rightarrow_{t_n} c_n) \in \text{Path}$, let $pr_{\mathfrak{S}, \sigma_0}(f) = pr_{\mathfrak{S}, \sigma_0}(c_0) \cdot pr_{\mathfrak{S}}(c_0 \rightarrow_{t_1} c_1) \cdot \dots \cdot pr_{\mathfrak{S}}(c_{n-1} \rightarrow_{t_n} c_n)$. Here, all paths f which are not “admissible” (e.g., guards are not fulfilled, transitions are starting or ending in wrong locations, etc.) have probability $pr_{\mathfrak{S}, \sigma_0}(f) = 0$.

The semantics of PIPs can be defined via a corresponding probability space, obtained by a standard cylinder construction. Let $\mathbb{P}_{\mathfrak{S},\sigma_0}$ denote the probability measure which lifts $pr_{\mathfrak{S},\sigma_0}$ to cylinder sets: For any $f \in \text{Path}$, we have $pr_{\mathfrak{S},\sigma_0}(f) = \mathbb{P}_{\mathfrak{S},\sigma_0}(\text{Pre}_f)$ for the set Pre_f of all infinite runs with prefix f . So $\mathbb{P}_{\mathfrak{S},\sigma_0}(\Theta)$ is the probability that a run from $\Theta \subseteq \text{Run}$ is obtained when using the scheduler \mathfrak{S} and starting in σ_0 . Let $\mathbb{E}_{\mathfrak{S},\sigma_0}$ denote the associated expected value operator. So for any random variable $X : \text{Run} \rightarrow \overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, we have $\mathbb{E}_{\mathfrak{S},\sigma_0}(X) = \sum_{n \in \overline{\mathbb{N}}} n \cdot \mathbb{P}_{\mathfrak{S},\sigma_0}(X = n)$. For a detailed construction, see [26].

Definition 2 (Expected Runtime). For $g \in \mathcal{GT}$, $\mathcal{R}_g : \text{Run} \rightarrow \overline{\mathbb{N}}$ is a random variable with $\mathcal{R}_g(c_0 \rightarrow_{t_1} c_1 \rightarrow_{t_2} \dots) = |\{i \in \mathbb{N} \mid t_i \in g\}|$, i.e., $\mathcal{R}_g(\vartheta)$ is the number of times that a transition from g was applied in the run $\vartheta \in \text{Run}$. Moreover, the random variable $\mathcal{R} : \text{Run} \rightarrow \overline{\mathbb{N}}$ denotes the number of transitions that were executed before termination, i.e., for all $\vartheta \in \text{Run}$ we have $\mathcal{R}(\vartheta) = \sum_{g \in \mathcal{GT}} \mathcal{R}_g(\vartheta)$. For a scheduler \mathfrak{S} and $\sigma_0 \in \Sigma$, the expected runtime of g is $\mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R}_g)$ and the expected runtime of the program is $\mathcal{R}_{\mathfrak{S},\sigma_0} = \mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R})$.

The goal of complexity analysis for a PIP is to compute a bound on its *expected runtime complexity*. The set of *bounds* \mathcal{B} consists of all functions from $\Sigma \rightarrow \mathbb{R}_{\geq 0}$.

Definition 3 (Expected Runtime Bound and Complexity [27]). The function $\mathcal{RB} : \mathcal{GT} \rightarrow \mathcal{B}$ is an *expected runtime bound* if $(\mathcal{RB}(g))(\sigma_0) \geq \sup_{\mathfrak{S}} \mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R}_g)$ for all $\sigma_0 \in \Sigma$ and all $g \in \mathcal{GT}$. Then $\sum_{g \in \mathcal{GT}} \mathcal{RB}(g)$ is a *bound on the expected runtime complexity of the whole program*, i.e., $\sum_{g \in \mathcal{GT}} ((\mathcal{RB}(g))(\sigma_0)) \geq \sup_{\mathfrak{S}} \mathcal{R}_{\mathfrak{S},\sigma_0}$ for all $\sigma_0 \in \Sigma$.

3 Control-Flow Refinement for PIPs

We now introduce our novel CFR algorithm for *probabilistic* integer programs, based on the partial evaluation technique for non-probabilistic programs from [13, 14, 18]. In particular, our algorithm coincides with the classical CFR technique when the program is non-probabilistic. The goal of CFR is to transform a program \mathcal{P} into a program \mathcal{P}' which is “easier” to analyze. Thm. 4 shows the soundness of our approach, i.e., that \mathcal{P} and \mathcal{P}' have the same expected runtime complexity.

Our CFR technique considers “abstract” evaluations which operate on sets of states. These sets are characterized by conjunctions τ of constraints from $\mathcal{C}(\mathcal{PV})$, i.e., τ stands for all states $\sigma \in \Sigma$ with $\sigma \models \tau$. We now label locations ℓ by formulas τ which describe (a superset of) those states σ which can occur in ℓ , i.e., where a configuration (ℓ, σ) is reachable from some initial configuration (ℓ_0, σ_0) . We begin with labeling every location by the constraint **true**. Then we add new copies of the locations with refined labels τ by considering how the updates of transitions affect the constraints of their start locations and their guards. The labeled locations become the new locations in the refined program.

Since a location might be reachable by different paths, we may construct several variants $\langle \ell, \tau_1 \rangle, \dots, \langle \ell, \tau_n \rangle$ of the same original location ℓ . Thus, the formulas τ are not necessarily invariants that hold for *all* evaluations that reach a location ℓ , but we perform a case analysis and split up a location ℓ according to the different sets of states that may reach ℓ . Our approach ensures that a labeled location $\langle \ell, \tau \rangle$ can only be reached by configurations (ℓ, σ) where $\sigma \models \tau$.

We apply CFR only *on-demand* on a (sub)set of transitions $\mathcal{S} \subseteq \mathcal{T}$ (thus, CFR can be performed in a *modular* way for different subsets \mathcal{S}). In practice, we choose \mathcal{S} heuristically and use CFR only on transitions where our currently inferred runtime bounds are “not yet good enough”. Then, for $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$, the result of the CFR algorithm is the program $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \langle \ell_0, \mathbf{true} \rangle, \mathcal{GT}')$ where \mathcal{L}' and \mathcal{GT}' are the smallest sets satisfying the properties (3), (4), and (5) below.

First, we require that for all $\ell \in \mathcal{L}$, all “original” locations $\langle \ell, \mathbf{true} \rangle$ are in \mathcal{L}' . In these locations, we do not have any information on the possible states yet:

$$\forall \ell \in \mathcal{L}. \langle \ell, \mathbf{true} \rangle \in \mathcal{L}' \quad (3)$$

If we already introduced a location $\langle \ell, \tau \rangle \in \mathcal{L}'$ and there is a transition $(\ell, \varphi, p, \eta, \ell')$ in \mathcal{S} , then (4) requires that we also add the location $\langle \ell', \tau_{\varphi, \eta, \ell'} \rangle$ to \mathcal{L}' . The formula $\tau_{\varphi, \eta, \ell'}$ over-approximates the set of states that can result from states that satisfy τ and the guard φ of the transition when applying the update η . More precisely, $\tau_{\varphi, \eta, \ell'}$ has to satisfy $(\tau \wedge \varphi) \models \eta(\tau_{\varphi, \eta, \ell'})$. For example, if $\tau = (x = 0)$, $\varphi = \mathbf{true}$, and $\eta(x) = x - 1$, then we might have $\tau_{\varphi, \eta, \ell'} = (x = -1)$.

To ensure that every $\ell' \in \mathcal{L}$ only gives rise to *finitely* many new labeled locations $\langle \ell', \tau_{\varphi, \eta, \ell'} \rangle$, we perform *property-based abstraction*: For every location ℓ' , we use a finite so-called *abstraction layer* $\alpha_{\ell'} \subset \{p_1 \sim p_2 \mid p_1, p_2 \in \mathbb{Z}[\mathcal{PV}]\}$ and $\sim \in \{<, \leq, =\}$ (see [14] for heuristics to compute $\alpha_{\ell'}$). Then we require that $\tau_{\varphi, \eta, \ell'}$ must be a conjunction of constraints from $\alpha_{\ell'}$ (i.e., $\tau_{\varphi, \eta, \ell'} \subseteq \alpha_{\ell'}$ when regarding sets of constraints as their conjunction). This guarantees termination of our CFR algorithm, since for every location ℓ' there are only finitely many possible labels.

$$\forall \langle \ell, \tau \rangle \in \mathcal{L}'. \forall (\ell, \varphi, p, \eta, \ell') \in \mathcal{S}. \langle \ell', \tau_{\varphi, \eta, \ell'} \rangle \in \mathcal{L}' \\ \text{where } \tau_{\varphi, \eta, \ell'} = \{\psi \in \alpha_{\ell'} \mid (\tau \wedge \varphi) \models \eta(\psi)\} \quad (4)$$

Finally, we have to ensure that \mathcal{GT}' contains all “necessary” (general) transitions. To this end, we consider all $g \in \mathcal{GT}$. The transitions $(\ell, \varphi, p, \eta, \ell')$ in $g \cap \mathcal{S}$ now have to connect the appropriately labeled locations. Thus, for all labeled variants $\langle \ell, \tau \rangle \in \mathcal{L}'$, we add the transition $(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \tau_{\varphi, \eta, \ell'} \rangle)$. In contrast, the transitions $(\ell, \varphi, p, \eta, \ell')$ in $g \setminus \mathcal{S}$ only reach the location where ℓ' is labeled with \mathbf{true} , i.e., here we add the transition $(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \mathbf{true} \rangle)$.

$$\forall \langle \ell, \tau \rangle \in \mathcal{L}'. \forall g \in \mathcal{GT}. \\ \{(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \tau_{\varphi, \eta, \ell'} \rangle) \mid (\ell, \varphi, p, \eta, \ell') \in g \cap \mathcal{S}\} \cup \\ \{(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \mathbf{true} \rangle) \mid (\ell, \varphi, p, \eta, \ell') \in g \setminus \mathcal{S}\} \in \mathcal{GT}' \quad (5)$$

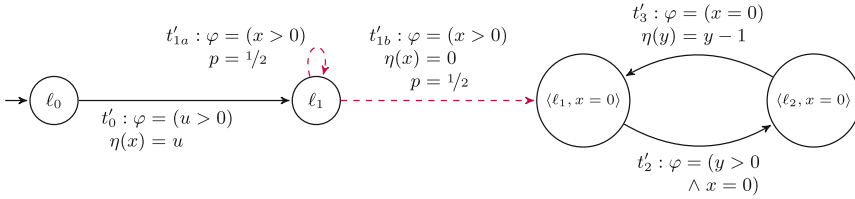


Fig. 2. Result of Control-Flow Refinement with $\mathcal{S} = \{t_{1a}, t_{1b}, t_2, t_3\}$

\mathcal{L}' and $\bigcup_{g \in \mathcal{G}T'} g$ are finite due to the property-based abstraction, as there are only finitely many possible labels for each location. Hence, repeatedly “unrolling” transitions by (5) leads to the (unique) least fixpoint. Moreover, (5) yields proper general transitions, i.e., their probabilities still add up to 1. In practice, we remove transitions with unsatisfiable guards, and locations that are not reachable from $\langle \ell_0, \text{true} \rangle$. Thm. 4 shows the soundness of our approach (see [26] for its proof).

Theorem 4 (Soundness of CFR for PIPs). *Let $\mathcal{P}' = (\mathcal{P}\mathcal{V}, \mathcal{L}', \langle \ell_0, \text{true} \rangle, \mathcal{G}T')$ be the PIP such that \mathcal{L}' and $\mathcal{G}T'$ are the smallest sets satisfying (3), (4), and (5). Let $\mathcal{R}_{\mathcal{S}, \sigma_0}^{\mathcal{P}}$ and $\mathcal{R}_{\mathcal{S}, \sigma_0}^{\mathcal{P}'}$ be the expected runtimes of \mathcal{P} and \mathcal{P}' , respectively. Then for all $\sigma_0 \in \Sigma$ we have $\sup_{\mathcal{S}} \mathcal{R}_{\mathcal{S}, \sigma_0}^{\mathcal{P}} = \sup_{\mathcal{S}} \mathcal{R}_{\mathcal{S}, \sigma_0}^{\mathcal{P}'}$.*

CFR Algorithm and its Runtime: To implement the fixpoint construction of Thm. 4 (i.e., to compute the PIP \mathcal{P}'), our algorithm starts by introducing all “original” locations $\langle \ell, \text{true} \rangle$ for $\ell \in \mathcal{L}$ according to (3). Then it iterates over all labeled locations $\langle \ell, \tau \rangle$ and all transitions $t \in \mathcal{T}$. If the start location of t is ℓ , then the algorithm extends $\mathcal{G}T'$ by a new transition according to (5). Moreover, it also adds the corresponding labeled target location to \mathcal{L}' (as in (4)), if \mathcal{L}' did not contain this labeled location yet. Afterwards, we mark $\langle \ell, \tau \rangle$ as finished and proceed with a previously computed labeled location that is not marked yet. So our implementation iteratively “unrolls” transitions by (5) until no new labeled locations are obtained (this yields the least fixpoint mentioned above). Thus, unrolling steps with transitions from $\mathcal{T} \setminus \mathcal{S}$ do not invoke further computations.

To over-approximate the runtime of this algorithm, note that for every location $\ell \in \mathcal{L}$, there can be at most $2^{|\alpha_{\ell}|}$ many labeled locations of the form $\langle \ell, \tau \rangle$. So if $\mathcal{L} = \{\ell_0, \dots, \ell_n\}$, then the overall number of labeled locations is at most $2^{|\alpha_{\ell_0}|} + \dots + 2^{|\alpha_{\ell_n}|}$. Hence, the algorithm performs at most $|\mathcal{T}| \cdot (2^{|\alpha_{\ell_0}|} + \dots + 2^{|\alpha_{\ell_n}|})$ unrolling steps.

Example 5. For the PIP in Fig. 1 and $\mathcal{S} = \{t_{1a}, t_{1b}, t_2, t_3\}$, by (3) we start with $\mathcal{L}' = \{\langle \ell_i, \text{true} \rangle \mid i \in \{0, 1, 2\}\}$. We abbreviate $\langle \ell_i, \text{true} \rangle$ by ℓ_i in the final result of the CFR algorithm in Fig. 2. As $t_0 \in \{t_0\} \setminus \mathcal{S}$, by (5) t_0 is redirected such that it starts at $\langle \ell_0, \text{true} \rangle$ and ends in $\langle \ell_1, \text{true} \rangle$, resulting in t'_0 . We always use primes to indicate the correspondence between new and original transitions.

Next, we consider $\{t_{1a}, t_{1b}\} \subseteq \mathcal{S}$ with the guard $\varphi = (x > 0)$ and start location $\langle \ell_1, \text{true} \rangle$. We first handle t_{1a} which has the update $\eta = \text{id}$. We use the

abstraction layer $\alpha_{\ell_0} = \emptyset$, $\alpha_{\ell_1} = \{x = 0\}$, and $\alpha_{\ell_2} = \{x = 0\}$. Thus, we have to find all $\psi \in \alpha_{\ell_1} = \{x = 0\}$ such that $(\mathbf{true} \wedge x > 0) \models \eta(\psi)$. Hence, $\tau_{x>0, \text{id}, \ell_1}$ is the empty conjunction \mathbf{true} as no ψ from α_{ℓ_1} satisfies this property. We obtain

$$t'_{1a} : (\langle \ell_1, \mathbf{true} \rangle, x > 0, 1/2, \text{id}, \langle \ell_1, \mathbf{true} \rangle).$$

In contrast, t_{1b} has the update $\eta(x) = 0$. To determine $\tau_{x>0, \eta, \ell_1}$, again we have to find all $\psi \in \alpha_{\ell_1} = \{x = 0\}$ such that $(\mathbf{true} \wedge x > 0) \models \eta(\psi)$. Here, we get $\tau_{x>0, \eta, \ell_1} = (x = 0)$. Thus, by (4) we create the location $\langle \ell_1, x = 0 \rangle$ and obtain

$$t'_{1b} : (\langle \ell_1, \mathbf{true} \rangle, x > 0, 1/2, \eta(x) = 0, \langle \ell_1, x = 0 \rangle).$$

As t_{1a} and t_{1b} form one general transition, by (5) we obtain $\{t'_{1a}, t'_{1b}\} \in \mathcal{GT}'$.

Now, we consider transitions resulting from $\{t_{1a}, t_{1b}\}$ with the start location $\langle \ell_1, x = 0 \rangle$. However, $\tau = (x = 0)$ and the guard $\varphi = (x > 0)$ are conflicting, i.e., the transitions would have an unsatisfiable guard $\tau \wedge \varphi$ and are thus omitted.

Next, we consider transitions resulting from t_2 with $\langle \ell_1, \mathbf{true} \rangle$ or $\langle \ell_1, x = 0 \rangle$ as their start location. Here, we obtain two (general) transitions $\{t'_2\}, \{t''_2\} \in \mathcal{GT}'$:

$$\begin{aligned} t'_2 &: (\langle \ell_1, x = 0 \rangle, y > 0 \wedge x = 0, 1, \text{id}, \langle \ell_2, x = 0 \rangle) \\ t''_2 &: (\langle \ell_1, \mathbf{true} \rangle, y > 0 \wedge x = 0, 1, \text{id}, \langle \ell_2, x = 0 \rangle) \end{aligned}$$

However, t''_2 can be ignored since $x = 0$ contradicts the invariant $x > 0$ at $\langle \ell_1, \mathbf{true} \rangle$. KoAT uses Apron [20] to infer invariants like $x > 0$ automatically. Finally, t_3 leads to the transition $t'_3 : (\langle \ell_2, x = 0 \rangle, x = 0, 1, \eta(y) = y - 1, \langle \ell_1, x = 0 \rangle)$. Thus, we obtain $\mathcal{L}' = \{\langle \ell_i, \mathbf{true} \rangle \mid i \in \{0, 1\}\} \cup \{\langle \ell_i, x = 0 \rangle \mid i \in \{1, 2\}\}$.

KoAT infers a bound $\mathcal{RB}(g)$ for each $g \in \mathcal{GT}$ individually (thus, non-probabilistic program parts can be analyzed by classical techniques). Then $\sum_{g \in \mathcal{GT}} \mathcal{RB}(g)$ is a bound on the expected runtime complexity of the whole program, see Definition 3.

Example 6. We now infer a bound on the expected runtime complexity of the PIP in Fig. 2. Transition t'_0 is not on a cycle, i.e., it can be evaluated at most once. So $\mathcal{RB}(\{t'_0\}) = 1$ is an (expected) runtime bound for the general transition $\{t'_0\}$.

For the general transition $\{t'_{1a}, t'_{1a}\}$, KoAT infers the expected runtime bound 2 via probabilistic linear ranking functions (PLRFs, see e.g., [27]). More precisely, KoAT finds the *constant* PLRF $\{\ell_1 \mapsto 2, \langle \ell_1, x = 0 \rangle \mapsto 0\}$. In contrast, in the original program of Fig. 1, $\{t_{1a}, t_{1b}\}$ is not decreasing w.r.t. any constant PLRF, because t_{1a} and t_{1b} have the same target location. So here, every PLRF where $\{t_{1a}, t_{1b}\}$ decreases in expectation depends on x . However, such PLRFs do not yield a finite runtime bound in the end, as t_0 instantiates x by the non-deterministic value u . Therefore, KoAT fails on the program of Fig. 1 without using CFR.

For the program of Fig. 2, KoAT infers $\mathcal{RB}(\{t'_2\}) = \mathcal{RB}(\{t'_3\}) = y$. By adding all runtime bounds, we obtain the bound $3 + 2 \cdot y$ on the expected runtime complexity of the program in Fig. 2 and thus by Theorem 4 also of the program in Fig. 1.

Table 1. Evaluation of CFR on Probabilistic Programs

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \omega$	AVG ⁺ (s)	AVG(s)
KoAT+CFR	11 (2)	56 (12)	14	2	1	84 (14)	11.68	11.34
KoAT	9	41 (1)	16 (1)	2	1	69 (2)	2.71	2.41
Absynth	7	35	9	0	0	51	2.86	37.48
eco-imp	8	35	6	0	0	49	0.34	68.02

4 Implementation, Evaluation, and Conclusion

We presented a novel control-flow refinement technique for probabilistic programs and proved that it does not modify the program’s expected runtime complexity. This allows us to combine CFR with approaches for complexity analysis of probabilistic programs. Compared to its variant for non-probabilistic programs, the soundness proof of Theorem 4 for probabilistic programs is considerably more involved.

Up to now, our complexity analyzer KoAT used the tool iRankFinder [13] for CFR of non-probabilistic programs [18]. To demonstrate the benefits of CFR for complexity analysis of probabilistic programs, we now replaced the call to iRankFinder in KoAT by a native implementation of our new CFR algorithm. KoAT is written in OCaml and it uses Z3 [12] for SMT solving, Apron [20] to generate invariants, and the Parma Polyhedra Library [8] for computations with polyhedra.

We used all 75 probabilistic benchmarks from [27,29] and added 15 new benchmarks including our leading example and problems adapted from the *Termination Problem Data Base* [33], e.g., a probabilistic version of McCarthy’s 91 function. Our benchmarks also contain examples where CFR is useful even if it cannot separate probabilistic from non-probabilistic program parts as in our leading example.

Table 1 shows the results of our experiments. We compared the configuration of KoAT with CFR (“KoAT+CFR”) against KoAT without CFR. Moreover, as in [27], we also compared with the main other recent tools for inferring upper bounds on the expected runtimes of probabilistic integer programs (Absynth [29] and eco-imp [7]). As in the *Termination Competition* [17], we used a timeout of 5 min per example. The first entry in every cell is the number of benchmarks for which the tool inferred the respective bound. In brackets, we give the corresponding number when only regarding our new examples. For example, KoAT+CFR finds a finite expected runtime bound for 84 of the 90 examples. A linear expected bound (i.e., in $\mathcal{O}(n)$) is found for 56 of these 84 examples, where 12 of these benchmarks are from our new set. AVG(s) is the average runtime in seconds on all benchmarks and AVG⁺(s) is the average runtime on all successful runs.

The experiments show that similar to its benefits for non-probabilistic programs [18], CFR also increases the power of automated complexity analysis

for probabilistic programs substantially, while the runtime of the analyzer may become longer since CFR increases the size of the program. The experiments also indicate that a related CFR technique is not available in the other complexity analyzers. Thus, we conjecture that other tools for complexity or termination analysis of PIPs would also benefit from the integration of our CFR technique.

KoAT's source code, a binary, and a Docker image are available at:

https://koat.verify.rwth-aachen.de/prob_cfr

The website also explains how to use our CFR implementation separately (without the rest of KoAT), in order to access it as a black box by other tools. Moreover, the website provides a *web interface* to directly run KoAT online, and details on our experiments, including our benchmark collection.

Acknowledgements. We thank Yoann Kehler for helping with the implementation of our CFR technique in KoAT.

References

1. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs'. In: Proceedings of the ACM on Programming Languages, vol. 2. POPL (2017). <https://doi.org/10.1145/3158122>
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. **413**(1), 142–159 (2012). <https://doi.org/10.1016/j.tcs.2011.07.009>
4. Albert, E., Bofill, M., Borralleras, C., Martín-Martín, E., Rubio, A.: Resource analysis driven by (conditional) termination proofs. Theory Pract. Log. Program. **19**(5–6), 722–739 (2019). <https://doi.org/10.1017/S1471068419000152>
5. Alias, C., Darté, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
6. Avanzini, M., Moser, G.: A combination framework for complexity. In: van Raamsdonk, F. (ed.) RTA 2013. LIPIcs, vol. 21, pp. 55–70 (2013). <https://doi.org/10.4230/LIPIcs.RTA.2013.55>
7. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. In: Proceedings of the ACM on Programming Languages, vol. 4. OOPSLA (2020). <https://doi.org/10.1145/3428240>
8. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**, 3–21 (2008). <https://doi.org/10.1016/j.scico.2007.08.001>

9. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C., Verscht, L.: A calculus for amortized expected runtimes. In: Proceedings of the ACM on Programming Languages, vol. 7. POPL (2023). <https://doi.org/10.1145/3571260>
10. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.* **38**, 1–50 (2016). <https://doi.org/10.1145/2866575>
11. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Grove, D., Blackburn, S.M. (eds.) PLDI 2015, pp. 467–478 (2015). <https://doi.org/10.1145/2737924.2737955>
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Doménech, J.J., Genaim, S.: iRankFinder. In: Lucas, S. (ed.) WST 2018, p. 83 (2018). <http://wst2018.webs.upv.es/wst2018proceedings.pdf>
14. Doménech, J.J., Gallagher, J.P., Genaim, S.: Control-flow refinement by partial evaluation, and its application to termination and cost analysis. In: Theory and Practice of Logic Programming vol. 19(5-6), pp. 990–1005 (2019). <https://doi.org/10.1017/S1471068419000310>
15. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
16. Frohn, F., Giesl, J.: Complexity analysis for Java with AProVE. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 85–101. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_6
17. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 156–166. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_10
18. Giesl, J., Lommen, N., Hark, M., Meyer, F.: Improving automatic complexity analysis of integer programs. In: The Logic of Software. A Tasting Menu of Formal Methods. LNCS 13360, pp. 193–228 (2022). https://doi.org/10.1007/978-3-031-08166-8_10
19. Hoffmann, J., Das, A., Weng, S.-C.: Towards automatic resource bound analysis for OCaml’. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017, pp. 359–373 (2017). <https://doi.org/10.1145/3009837.3009842>
20. Jeannet, B., Miné, A.: APRON: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
21. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM* **65**, 1–68 (2018). <https://doi.org/10.1145/3208102>
22. Kaminski, B.L., Katoen, J.-P., Matheja, C.: Expected runtime analysis by program verification. In: Barthe, G., Katoen, J.-P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 185–220. Cambridge University Press (2020). <https://doi.org/10.1017/9781108770750.007>
23. Leutgeb, L., Moser, G., Zuleger, F.: Automated expected amortised cost analysis of probabilistic data structures. In: Shoham, S., Vizel, Y. (eds.) CAV 2022. LNCS, vol. 13372, pp. 70–91 (2022). https://doi.org/10.1007/978-3-031-13188-2_4

24. Lommen, N., Meyer, F., Giesl, J.: Automatic complexity analysis of integer programs via triangular weakly non-linear loops. In: Blanchette, J., Kovács, L., Paton, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 734–754 (2022). https://doi.org/10.1007/978-3-031-10769-6_43
25. Lommen, N., Giesl, J.: Targeting completeness: using closed forms for size bounds of integer programs. In: Sattler, U., Suda, M. (eds.) FroCoS 2023. LNCS, vol. 14279, pp. 3–22 (2023). https://doi.org/10.1007/978-3-031-43369-6_1
26. Lommen, N., Meyer, E., Giesl, J.: Control-flow refinement for complexity analysis of probabilistic programs in KoAT. <https://doi.org/10.48550/arXiv.2402.03891>. Corr abs/2402.03891(2024)
27. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
28. Moser, G., Schaper, M.: From JINJA bytecode to term rewriting: a complexity reflecting transformation. *Inf. Comput.* **261**, 116–143 (2018). <https://doi.org/10.1016/j.ic.2018.05.007>
29. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018, pp. 496–512 (2018). <https://doi.org/10.1145/3192366.3192394>
30. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. *J. Autom. Reason.* **51**, 27–56 (2013). <https://doi.org/10.1007/s10817-013-9277-6>
31. Schröer, P., Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: A deductive verification infrastructure for probabilistic programs. In: Proceedings of the ACM on Programming Languages, vol. 7. OOPSLA, pp. 2052–2082 (2023). <https://doi.org/10.1145/3622870>
32. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reason.* **59**(1), 3–45 (2017). <https://doi.org/10.1007/s10817-016-9402-4>
33. TPDB (Termination Problem Data Base). <https://github.com/TermCOMP/TPDB>
34. Wang, D., Kahn, D.M., Hoffmann, J.: Raising expectations: automating expected cost analysis with types. In: Proceedings of the ACM on Programming Languages, vol. 4. ICFP (2020). <https://doi.org/10.1145/3408992>

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

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





On the (In-)Completeness of Destructive Equality Resolution in the Superposition Calculus

Uwe Waldmann^(✉) 

MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
uwe@mpi-inf.mpg.de

Abstract. Bachmair’s and Ganzinger’s abstract redundancy concept for the Superposition Calculus justifies almost all operations that are used in superposition provers to delete or simplify clauses, and thus to keep the clause set manageable. Typical examples are tautology deletion, subsumption deletion, and demodulation, and with a more refined definition of redundancy joinability and connectedness can be covered as well. The notable exception is DESTRUCTIVE EQUALITY RESOLUTION, that is, the replacement of a clause $x \approx t \vee C$ with $x \notin \text{vars}(t)$ by $C\{x \mapsto t\}$. This operation is implemented in state-of-the-art provers, and it is clearly useful in practice, but little is known about how it affects refutational completeness. We demonstrate on the one hand that the naive addition of DESTRUCTIVE EQUALITY RESOLUTION to the standard abstract redundancy concept renders the calculus refutationally incomplete. On the other hand, we present several restricted variants of the Superposition Calculus that are refutationally complete even with DESTRUCTIVE EQUALITY RESOLUTION.

Keywords: Automated theorem proving · First-order logic · Superposition calculus

1 Introduction

Bachmair’s and Ganzinger’s Superposition Calculus [2] comes with an abstract redundancy concept that describes under which circumstances clauses can be simplified away or deleted during a saturation without destroying the refutational completeness of the calculus. Typical concrete simplification and deletion techniques that are justified by the abstract redundancy concept are tautology deletion, subsumption deletion, and demodulation, and with a more refined definition of redundancy (Duarte and Korovin [4]) joinability and connectedness can be covered as well.

There is one simplification technique left that is not justified by Bachmair’s and Ganzinger’s redundancy criterion, namely DESTRUCTIVE EQUALITY RESOLUTION (DER), that is, the replacement of a clause $x \approx t \vee C$ with $x \notin \text{vars}(t)$ by $C\{x \mapsto t\}$. This operation is for instance implemented in the E prover (Schulz

[6]), and it has been shown to be useful in practice: It increases the number of problems that E can solve and it also reduces E's runtime per solved problem. The question how it affects the refutational completeness of the calculus, both in theory and in practice, has been open, though (except for the special case that t is also a variable, where DER is equivalent to selecting the literal $x \not\approx t$ so that EQUALITY RESOLUTION becomes the only possible inference with this clause).

In this paper we demonstrate on the one hand that the naive addition of DER to the standard abstract redundancy concept renders the calculus refutationally incomplete. On the other hand, we present several restricted variants of the Superposition Calculus that are refutationally complete even with DER.

By lack of space, some proofs had to be omitted from this version of the paper; they can be found in the technical report [7].

2 Preliminaries

Basic Notions. We refer to (Baader and Nipkow [1]) for basic notations and results on orderings, multiset operations, and term rewriting.

We use standard set operation symbols like \cup and \in and curly braces also for finite multisets. The union $S \cup S'$ of the multisets S and S' over some set M is defined by $(S \cup S')(x) = S(x) + S'(x)$ for every $x \in M$.

Without loss of generality we assume that all most general unifiers that we consider are idempotent. Note that if σ is an idempotent most general unifier and θ is a unifier then $\theta \circ \sigma = \theta$.

A clause is a finite multiset of equational literals $s \approx t$ or $s \not\approx t$, written as a disjunction. The empty clause is denoted by \perp . We call a literal L in a clause $C \vee L$ maximal w.r.t. a strict literal ordering, if there is no literal in C that is larger than L ; we call it strictly maximal, if there is no literal in C that is larger than or equal to L .

We write a rewrite rule as $u \rightarrow v$. Semantically, a rule $u \rightarrow v$ is equivalent to an equation $u \approx v$. If R is a rewrite system, that is, a set of rewrite rules, we write $s \rightarrow_R t$ to indicate that the term s can be reduced to the term t by applying a rule from R . A rewrite system is called left-reduced, if there is no rule $u \rightarrow v \in R$ such that u can be reduced by a rule from $R \setminus \{u \rightarrow v\}$.

The Superposition Calculus. We summarize the key elements of Bachmair's and Ganzinger's Superposition Calculus [2].

Let \succ be a reduction ordering that is total on ground terms. We extend \succ to an ordering on literals, denoted by \succ_L ,¹ by mapping positive literals $s \approx t$ to multisets $\{s, t\}$ and negative literals $s \not\approx t$ to multisets $\{s, s, t, t\}$ and by comparing the resulting multisets using the multiset extension of \succ . We extend the literal ordering \succ_L to an ordering on clauses, denoted by \succ_C , by comparing the multisets of literals in these clauses using the multiset extension of \succ_L .

¹ There are several equivalent ways to define \succ_L .

The inference system of the Superposition Calculus consists of the rules SUPERPOSITION, EQUALITY RESOLUTION, and EQUALITY FACTORING.²

$$\text{SUPERPOSITION:} \quad \frac{D' \vee t \approx t' \quad C' \vee L[u]}{(D' \vee C' \vee L[t'])\sigma}$$

where u is not a variable; $\sigma = \text{mgu}(t, u)$; $(C' \vee L[u])\sigma \not\leq_C (D' \vee t \approx t')\sigma$; $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$; either $L[u]$ is a positive literal $s[u] \approx s'$ and $L[u]\sigma$ is strictly maximal in $(C' \vee L[u])\sigma$, or $L[u]$ is a negative literal $s[u] \not\approx s'$ and $L[u]\sigma$ is maximal in $(C' \vee L[u])\sigma$; $t\sigma \not\leq t'\sigma$; and $s\sigma \not\leq s'\sigma$.

$$\text{EQUALITY RESOLUTION:} \quad \frac{C' \vee s \not\approx s'}{C'\sigma}$$

where $\sigma = \text{mgu}(s, s')$ and $(s \not\approx s')\sigma$ is maximal in $(C' \vee s \not\approx s')\sigma$.

$$\text{EQUALITY FACTORING:} \quad \frac{C' \vee r \approx r' \vee s \approx s'}{(C' \vee s' \not\approx r' \vee r \approx r')\sigma}$$

where $\sigma = \text{mgu}(s, r)$; $s\sigma \not\leq s'\sigma$; and $(s \approx s')\sigma$ is maximal in $(C' \vee r \approx r' \vee s \approx s')\sigma$.

The ordering restrictions can be overridden using *selection functions* that determine for each clause a subset of the negative literals that are available for inferences. For simplicity, we leave out this refinement in the rest of this paper. We emphasize, however, that all results that we present here hold also in the presence of selection functions; the required modifications of the proofs are straightforward.

A ground clause C is called (classically) redundant w.r.t. a set of ground clauses N , if it follows from clauses in N that are smaller than C w.r.t. \succ_C . A clause is called (classically) redundant w.r.t. a set of clauses N , if all its ground instances are redundant w.r.t. the set of ground instances of clauses in N .³ A ground inference with conclusion C_0 and right (or only) premise C is called redundant w.r.t. a set of ground clauses N , if one of its premises is redundant w.r.t. N , or if C_0 follows from clauses in N that are smaller than C . An inference is called redundant w.r.t. a set of clauses N , if all its ground instances are redundant w.r.t. the set of ground instances of clauses in N .

Redundancy of clauses and inferences as defined above is a redundancy criterion in the sense of (Waldmann et al. [8]). It justifies typical deletion and simplification techniques such as the deletion of tautological clauses, subsumption deletion (i.e., the deletion of a clause $C\sigma \vee D$ in the presence of a clause C) or demodulation (i.e., the replacement of a clause $C[s\sigma]$ by $C[t\sigma]$ in the presence of a unit clause $s \approx t$, provided that $s\sigma \succ t\sigma$).

² The EQUALITY FACTORING rule can be replaced by the ORDERED FACTORING and the MERGING PARAMODULATION rule. Our results hold also for this variant.

³ Note that “redundancy” is called “compositeness” in Bachmair and Ganzinger’s *J. Log. Comput.* article [2]. In later papers the standard terminology has changed.

3 Incompleteness

There are two special cases where DESTRUCTIVE EQUALITY RESOLUTION (DER) is justified by the classical redundancy criterion: First, if t is the smallest constant in the signature, then every ground instance $(x \not\approx t \vee C)\theta$ follows from the smaller ground instance $C\{x \mapsto t\}\theta$. Second, if t is another variable y , then every ground instance $(x \not\approx y \vee C)\theta$ follows from the smaller ground instance $C\{x \mapsto y\}\{y \mapsto s\}\theta$, where s is the smaller of $x\theta$ and $y\theta$.

But it is easy to see that this does not work in general: Let \succ be a Knuth-Bendix ordering with weights $w(f) = w(b) = 2$, $w(c) = w(d) = 1$, $w(z) = 1$ for all variables z , and let C be the clause $x \not\approx b \vee f(x) \approx d$. Then DER applied to C yields $D = f(b) \approx d$. Now consider the substitution $\theta = \{x \mapsto c\}$. The ground instance $C\theta = c \not\approx b \vee f(c) \approx d$ is a logical consequence of D , but since it is smaller than D itself, D makes neither $C\theta$ nor C redundant.

Moreover, the following example demonstrates that the Superposition Calculus becomes indeed incomplete, if we add DER as a simplification rule, i.e., if we extend the definition of redundancy in such a way that the conclusion of DER renders the premise redundant.

Example 1. Let \succ be a Knuth-Bendix ordering with weights $w(f) = 4$, $w(g) = 3$, $w(b) = 4$, $w(b') = 2$, $w(c) = w(c') = w(d) = 1$, $w(z) = 1$ for all variables z , and let N be the set of clauses

$$\begin{aligned} C_1 &= \underline{f(x, d)} \approx x \\ C_2 &= \underline{f(x, y)} \not\approx b \vee g(x) \approx d \\ C_3 &= \underline{b'} \approx c' \vee \underline{b} \approx c \\ C_4 &= \underline{g(b')} \not\approx g(c') \\ C_5 &= \underline{g(c)} \not\approx d \end{aligned}$$

where all the maximal terms in maximal literals are underlined.

At this point, neither demodulation nor subsumption is possible. The only inference that must be performed is SUPERPOSITION between C_1 and C_2 , yielding

$$C_6 = x \not\approx b \vee g(x) \approx d$$

and by using DER, C_6 is replaced by

$$C_7 = g(b) \approx d$$

We could now continue with a SUPERPOSITION between C_3 and C_7 , followed by a SUPERPOSITION with C_5 , followed by EQUALITY RESOLUTION, and obtain

$$C_8 = b' \approx c'$$

from which we can derive the empty clause by SUPERPOSITION with C_4 and once more by EQUALITY RESOLUTION. However, clause C_7 is in fact redundant: The

ground clauses C_3 and C_4 imply $b \approx c$; therefore C_7 follows from C_3 , C_4 , and the ground instances

$$\begin{aligned} C_1\{x \mapsto c\} &= f(c, d) \approx c \\ C_2\{x \mapsto c, y \mapsto d\} &= f(c, d) \not\approx b \vee g(c) \approx d \end{aligned}$$

Because all terms in these clauses are smaller than the maximal term $g(b)$ of C_7 , all the clauses are smaller than C_7 . Since C_7 is redundant, we are allowed to delete it, and then no further inferences are possible anymore. Therefore the clause set $N = \{C_1, \dots, C_5\}$ is saturated, even though it is inconsistent and does not contain the empty clause, which implies that the calculus is not refutationally complete anymore.

4 Completeness, Part I: The Horn Case

4.1 The Idea

On the one hand, Example 1 demonstrates that we cannot simply extend the standard redundancy criterion of the Superposition Calculus with DER without destroying refutational completeness, and that this holds even if we impose a particular strategy on simplification steps (say, that simplifications must be performed eagerly and that demodulation and subsumption have a higher precedence than DER). On the other hand, Example 1 is of course highly unrealistic: Even though clause C_7 is redundant w.r.t. the clauses C_1 , C_2 , C_3 , and C_4 , no reasonable superposition prover would ever detect this – in particular, since doing so would require to invent the instance $C_2\{x \mapsto c, y \mapsto d\}$ of C_2 , which is not in any way syntactically related to C_7 .⁴

This raises the question whether DER still destroys refutational completeness when we restrict the other deletion and simplification techniques to those that are typically implemented in superposition provers, such as tautology detection, demodulation, or subsumption. Are there alternative redundancy criteria that are refutationally complete together with the Superposition Calculus and that justify DER as well as (all/most) commonly implemented deletion and simplification techniques? Given the usual structure of the inductive completeness proofs for saturation calculi, developing such a redundancy criterion would mean in particular to find a suitable clause ordering with respect to which certain clauses have to be smaller than others. The following example illustrates a fundamental problem that we have to deal with:

Example 2. Let \succ be a Knuth-Bendix ordering with weights $w(f) = w(g) = w(h) = w(c) = 1$, $w(b) = 2$, $w(z) = 1$ for all variables z . Consider the following set of clauses:

⁴ In fact, a prover might use SMT-style heuristic grounding of non-ground clauses, but then finding the contradiction turns out to be easier than proving the redundancy of C_7 .

$$\begin{array}{ll}
 D_1 = h(x) \approx x & C_1 = h(x) \not\approx b \vee f(g(x)) \approx c \\
 & C_2 = x \not\approx b \vee f(g(x)) \approx c \\
 D_3 = h(c) \not\approx b \vee g(b) \approx g(c) & C_3 = f(g(b)) \approx c \\
 & C_4 = h(c) \not\approx b \vee f(g(c)) \approx c
 \end{array}$$

Demodulation of C_1 using D_1 yields C_2 , and if we want Demodulation to be a simplification, then every ground instance $C_1\theta$ should be larger than the corresponding ground instance $C_2\theta$ in the clause ordering.

DER of C_2 yields C_3 , and if we want DER to be a simplification, then every ground instance $C_2\theta$ should be larger than $C_3\theta = C_3$.

A SUPERPOSITION inference between D_3 and C_3 yields C_4 . The inductive completeness proof for the calculus relies on the fact that the conclusion of an inference is smaller than the largest premise, so C_3 should be larger than C_4 .

By transitivity we obtain that every ground instance $C_1\theta$ should be larger than C_4 in the clause ordering. The clause C_4 , however, *is* a ground instance of C_1 , which is clearly a contradiction.

On the other hand, a closer inspection reveals that, depending on the limit rewrite system R_* that is produced in the completeness proof for the Superposition Calculus, the SUPERPOSITION inference between D_3 and C_3 is only needed, when D_3 produces the rewrite rule $g(b) \rightarrow g(c) \in R_*$, and that the only critical case for DER is the one where b can be reduced by some rule in R_* . Since the limit rewrite system R_* is by construction left-reduced, these two conditions are mutually exclusive. This observation indicates that we might be able to find a suitable clause ordering if we choose it depending on R_* .

4.2 Ground Case

The Normalization Closure Ordering. Let \succ be a reduction ordering that is total on ground terms. Let R be a left-reduced ground rewrite system contained in \succ .

For technical reasons that will become clear later, we design our ground superposition calculus in such a way that it operates on ground closures $(C \cdot \theta)$. Logically, a ground closure $(C \cdot \theta)$ is equivalent to a ground instance $C\theta$, but an ordering may treat two closures that represent the same ground instance in different ways. We consider closures up to α -renaming and ignore the behavior of θ on variables that do not occur in C , that is, we treat closures $(C_1 \cdot \theta_1)$ and $(C_2 \cdot \theta_2)$ as equal whenever C_1 and C_2 are equal up to bijective variable renaming and $C_1\theta_1 = C_2\theta_2$. We also identify $(\perp \cdot \theta)$ and \perp .

Intuitively, in order to compare ground closures $C \cdot \theta$, we normalize all terms occurring in $C\theta$ with R , we compute the multiset of all the redexes occurring during the normalization and all the resulting normal forms, and we compare these multisets using the multiset extension of \succ . Since we would like to give redexes and normal forms in negative literals a slightly larger weight than redexes and normal forms in positive literals, and redexes in positive literals below the

top a slightly larger weight than redexes at the top, we combine each of these terms with a label (0 for positive at the top, 1 for positive below the top, 2 for negative). Moreover, whenever some term t occurs several times in C as a subterm, we want to count the redexes resulting from the normalization of $t\theta$ only once (with the maximum of the labels). The reason for this is that DER can produce several copies of the same term t in a clause if the variable to be eliminated occurs several times in the clause; by counting all redexes stemming from t only once, we ensure that this does not increase the total number of redexes. Formally, we first compute the set (not multiset!) of all subterms t of C , so that duplicates are deleted, and then compute the multiset of redexes for all terms $t\theta$ (and analogously for terms occurring at the top of a literal).

Definition 3. We define the subterm sets $ss_{>\epsilon}^+(C)$ and $ss^-(C)$ and the topterm sets $ts^+(C)$ and $ts^-(C)$ of a clause C by

$$\begin{aligned} ss^-(C) &= \{t \mid C = C' \vee s[t]_p \not\approx s'\} \\ ss_{>\epsilon}^+(C) &= \{t \mid C = C' \vee s[t]_p \approx s', p > \epsilon\} \\ ts^-(C) &= \{t \mid C = C' \vee t \not\approx t'\} \\ ts^+(C) &= \{t \mid C = C' \vee t \approx t'\}. \end{aligned}$$

We define the labeled subterm set $lss(C)$ and the labeled topterm set $lts(C)$ of a clause C by

$$\begin{aligned} lss(C) &= \{(t, 2) \mid t \in ss^-(C)\} \\ &\quad \cup \{(t, 1) \mid t \in ss_{>\epsilon}^+(C) \setminus ss^-(C)\} \\ &\quad \cup \{(t, 0) \mid t \in ts^+(C) \setminus (ss_{>\epsilon}^+(C) \cup ss^-(C))\} \\ lts(C) &= \{(t, 2) \mid t \in ts^-(C)\} \cup \{(t, 0) \mid t \in ts^+(C) \setminus ts^-(C)\}. \end{aligned}$$

We define the R -redex multiset $rm_R(t, m)$ of a labeled ground term (t, m) with $m \in \{0, 1, 2\}$ by

$$\begin{aligned} rm_R(t, m) &= \emptyset \text{ if } t \text{ is } R\text{-irreducible;} \\ rm_R(t, m) &= \{(u, m)\} \cup rm_R(t', m) \text{ if } t \rightarrow_R t' \text{ using the rule } u \rightarrow v \in R \\ &\quad \text{at position } p \text{ and } p = \epsilon \text{ or } m > 0; \\ rm_R(t, m) &= \{(u, 1)\} \cup rm_R(t', m) \text{ if } t \rightarrow_R t' \text{ using the rule } u \rightarrow v \in R \\ &\quad \text{at position } p \text{ and } p > \epsilon \text{ and } m = 0. \end{aligned}$$

Lemma 4. For every left-reduced ground rewrite system R contained in \succ , $rm_R(t, m)$ is well-defined.

Definition 5. We define the R -normalization multiset $nm_R(C \cdot \theta)$ of a ground closure $(C \cdot \theta)$ by

$$\begin{aligned} nm_R(C \cdot \theta) &= \bigcup_{(f(t_1, \dots, t_n), m) \in lss(C)} rm_R(f(t_1\theta \downarrow_R, \dots, t_n\theta \downarrow_R), m) \\ &\quad \cup \bigcup_{(x, m) \in lss(C)} rm_R(x\theta, m) \\ &\quad \cup \bigcup_{(t, m) \in lts(C)} \{(t\theta \downarrow_R, m)\} \end{aligned}$$

Example 6. Let $C = h(g(g(x))) \approx f(f(b))$; let $\theta = \{x \mapsto b\}$. Then $\text{lss}(C) = \{(h(g(g(x))), 0), (g(g(x)), 1), (g(x), 1), (x, 1), (f(f(b)), 0), (f(b), 1), (b, 1)\}$ and $\text{lts}(C) = \{(h(g(g(x))), 0), (f(f(b)), 0)\}$.

Let $R = \{f(b) \rightarrow b, g(g(b)) \rightarrow b\}$. Then $\text{nm}_R(C \cdot \theta) = \{(g(g(b)), 1), (f(b), 1), (f(b), 0), (h(b), 0), (b, 0)\}$, where the first element is a redex from the normalization of $g(g(x))\theta$, the second from the normalization of $f(b)\theta$, the third from the normalization of $f(f(b))\theta$. The remaining elements are the normal forms of $h(g(g(x)))\theta$ and $f(f(b))\theta$.

The R -normalization closure ordering \succ_R compares ground closures $(C \cdot \theta_1)$ and $(D \cdot \theta_2)$ using a lexicographic combination of three orderings:

- first, the multiset extension $((\succ, >)_{\text{lex}})_{\text{mul}}$ of the lexicographic combination of the reduction ordering \succ and the ordering $>$ on natural numbers applied to the multisets $\text{nm}_R(C \cdot \theta_1)$ and $\text{nm}_R(D \cdot \theta_2)$,
- second, the traditional clause ordering \succ_C applied to $C\theta_1$ and $D\theta_2$,
- and third, an arbitrary well-founded ordering \succ_{clo} on ground closures that is total on ground closures $(C \cdot \theta_1)$ and $(D \cdot \theta_2)$ with $C\theta_1 = D\theta_2$ and that has the property that $(C \cdot \theta_1) \succ_{\text{clo}} (D \cdot \theta_2)$ whenever $C\theta_1 = D\theta_2$ and D is an instance of C but not vice versa.

Lemma 7. *If $(C \cdot \theta)$ and $(C\sigma \cdot \theta')$ are ground closures, such that $C\theta = C\sigma\theta'$, and C and $C\sigma$ are not equal up to bijective renaming, then $(C \cdot \theta) \succ_R (C\sigma \cdot \theta')$.*

Example 8. Let $C = h(f(x)) \approx f(y)$, let $\theta' = \{x \mapsto b\}$; let $\theta = \{x \mapsto b, y \mapsto b\}$; let $\sigma = \{y \mapsto x\}$. Let $R = \{f(b) \rightarrow b\}$.

Then $\text{nm}_R(C \cdot \theta) = \{(f(b), 1), (f(b), 0), (h(b), 0), (b, 0)\}$ and $\text{nm}_R(C\sigma \cdot \theta') = \{(f(b), 1), (h(b), 0), (b, 0)\}$, and therefore $(C \cdot \theta) \succ_R (C\sigma \cdot \theta')$. The subterm $f(x)$ occurs twice in $C\sigma$ (with labels 0 and 1), but only once in $\text{lss}(C\sigma)$ (with the larger of the two labels), and the same holds for the redex $f(b)$ stemming from $f(x)\theta'$ in $\text{nm}_R(C\sigma \cdot \theta')$.

Parallel Superposition. In the normalization closure ordering, redexes and normal forms stemming from several occurrences of the same term u in a closure $(C \cdot \theta)$ are counted only once. When we perform a SUPERPOSITION inference, this fact leads to a small problem: Consider a closure $(C[u, u] \cdot \theta)$. In the R -normalization multiset of this closure, the redexes stemming from the two copies of $u\theta$ are counted only once. Now suppose that one of the two copies of u is replaced by a smaller term v in a SUPERPOSITION inference. The resulting closure $(C[v, u] \cdot \theta)$ should be smaller than the original one, but it isn't: The redexes stemming from $u\theta$ are still counted once, and additionally, the R -normalization multiset now contains the redexes stemming from $v\theta$.

There is an easy fix for this problem, though: We have to replace the ordinary SUPERPOSITION rule by a PARALLEL SUPERPOSITION rule, in which *all* copies of a term u in a clause C are replaced whenever one copy occurs in a maximal side of a maximal literal. Note that this is a well-known optimization that superposition provers implement (or should implement) anyhow.

We need one further modification of the inference rule: The side conditions of the superposition calculus use the traditional clause ordering \succ_c , but our completeness proof and redundancy criterion will be based on the orderings \succ_R . The difference between these orderings becomes relevant in particular when we consider (PARALLEL) SUPERPOSITION inferences where the clauses overlap at the top of a positive literal. In this case, the \succ_R -smaller of the two premises may actually be the \succ_c -larger one. Therefore, the usual condition that the left premise of a (PARALLEL) SUPERPOSITION inference has to be \succ_c -minimal has to be dropped for these inferences.

PARALLEL SUPERPOSITION:
$$\frac{D' \vee t \approx t' \quad C[u, \dots, u]_{p_1, \dots, p_k}}{(D' \vee C[t', \dots, t']_{p_1, \dots, p_k})\sigma}$$

where u is not a variable; $\sigma = \text{mgu}(t, u)$; p_1, \dots, p_k are all the occurrences of u in C ; if one of the occurrences of u in C is in a negative literal or below the top in a positive literal then $C\sigma \not\prec_c (D' \vee t \approx t')\sigma$; $(t \approx t')\sigma$ is strictly maximal in $(D' \vee t \approx t')\sigma$; either one of the occurrences of u in C is in a positive literal $L[u] = s[u] \approx s'$ and $L[u]\sigma$ is strictly maximal in $C\sigma$, or one of the occurrences of u in C is in a negative literal $L[u] = s[u] \not\approx s'$ and $L[u]\sigma$ is maximal in $(C' \vee L[u])\sigma$; $t\sigma \not\prec t'\sigma$; and $s\sigma \not\prec s'\sigma$.

Ground Closure Horn Superposition. We will show that our calculus is refutationally complete for Horn clauses by lifting a similar result for ground closure Horn superposition. We emphasize that our calculus is not a basic or constraint calculus such as (Bachmair et al. [3]) or (Nieuwenhuis and Rubio [5]). Even though the ground version that we present here operates on closures, it is essentially a rephrased version of the standard ground Superposition Calculus. This explains why we also have to consider superpositions below variable positions.

The ground closure calculus uses the following three inference rules. We assume that in binary inferences the variables in the premises $(D \cdot \theta_2)$ and $(C \cdot \theta_1)$ are renamed in such a way that C and D do not share variables. We can then assume without loss of generality that the substitutions θ_2 and θ_1 agree.

PARALLEL SUPERPOSITION I:
$$\frac{(D' \vee t \approx t' \cdot \theta) \quad (C[u, \dots, u]_{p_1, \dots, p_k} \cdot \theta)}{((D' \vee C[t', \dots, t']_{p_1, \dots, p_k})\sigma \cdot \theta)}$$

where u is not a variable; $t\theta = u\theta$; $\sigma = \text{mgu}(t, u)$; p_1, \dots, p_k are all the occurrences of u in C ; if one of the occurrences of u in C is in a negative literal or below the top in a positive literal then $(D' \vee t \approx t')\theta \prec_c C\theta$; one of the occurrences of u in C is either in a positive literal $s[u] \approx s'$ such that $(s[u] \approx s')\theta$ is strictly maximal in $C\theta$ or in a negative literal $s[u] \not\approx s'$ such that $(s[u] \not\approx s')\theta$ is maximal in $C\theta$; $s[u]\theta \succ s'\theta$; $(t \approx t')\theta$ is strictly maximal in $(D' \vee t \approx t')\theta$; and $t\theta \succ t'\theta$.

PARALLEL SUPERPOSITION II:
$$\frac{(D' \vee t \approx t' \cdot \theta) \quad (C \cdot \theta)}{(D' \vee C \cdot \theta[x \mapsto u[t'\theta]])}$$

where x is a variable of C ; $x\theta = u[t\theta]$; if one of the occurrences of x in C is in a negative literal or below the top in a positive literal then $(D' \vee t \approx t')\theta \prec_C C\theta$; one of the occurrences of x in C is either in a positive literal $s[x] \approx s'$ such that $(s[x] \approx s')\theta$ is strictly maximal in $C\theta$ or in a negative literal $s[x] \not\approx s'$ such that $(s[x] \approx s')\theta$ is maximal in $C\theta$; $s[x]\theta \succ s'\theta$; $(t \approx t')\theta$ is strictly maximal in $(D' \vee t \approx t')\theta$; and $t\theta \succ t'\theta$.

$$\text{EQUALITY RESOLUTION:} \quad \frac{(C' \vee s \not\approx s' \cdot \theta)}{(C' \sigma \cdot \theta)}$$

where $s\theta = s'\theta$; $\sigma = \text{mgu}(s, s')$; and $(s \not\approx s')\theta$ is maximal in $(C' \vee s \not\approx s')\theta$.

The following lemmas compare the conclusion $\text{concl}(\iota)$ of an inference ι with its right or only premise:

Lemma 9. *Let ι be a ground EQUALITY RESOLUTION inference. Then $\text{concl}(\iota)$ is \succ_R -smaller than its premise.*

Lemma 10. *Let ι be a ground PARALLEL SUPERPOSITION inference*

$$\frac{(D' \vee t \approx t' \cdot \theta) \quad (C[u, \dots, u]_{p_1, \dots, p_k} \cdot \theta)}{((D' \vee C[t', \dots, t']_{p_1, \dots, p_k}) \sigma \cdot \theta)}$$

with $t\theta = u\theta$ and $\sigma = \text{mgu}(t, u)$ or

$$\frac{(D' \vee t \approx t' \cdot \theta) \quad (C \cdot \theta)}{(D' \vee C \cdot \theta[x \mapsto u[t'\theta]])}$$

with $x\theta = u[t\theta]$. If $(t\theta \rightarrow t'\theta) \in R$, then $\text{concl}(\iota)$ is \succ_R -smaller than $(C \cdot \theta)$.

Proof. Since $t\theta$ is replaced by $t'\theta$ at all occurrences of u or at or below all occurrences of x in C , one copy of the redex $t\theta$ is removed from $\text{nm}_R(C \cdot \theta)$. Moreover all terms in $D'\theta$ are smaller than $t\theta$, and consequently all redexes stemming from $D'\theta$ are smaller than $t\theta$. Therefore $\text{nm}_R(C \cdot \theta)$ is larger than $\text{nm}_R(D' \vee C[t', \dots, t']_{p_1, \dots, p_k} \cdot \theta)$ or $\text{nm}_R(D' \vee C \cdot \theta[x \mapsto u[t'\theta]])$. In the second case, this implies $(C \cdot \theta) \succ_R \text{concl}(\iota)$ immediately. In the first case, it implies $(C \cdot \theta) \succ_R (D' \vee C[t', \dots, t']_{p_1, \dots, p_k} \cdot \theta)$ and $(C \cdot \theta) \succ_R \text{concl}(\iota)$ follows using Lemma 7. \square

Redundancy. We will now construct a redundancy criterion for ground closure Horn superposition that is based on the ordering(s) \succ_R .

Definition 11. Let N be a set of ground closures. A ground closure $(C \cdot \theta)$ is called redundant w.r.t. N , if for every left-reduced ground rewrite system R contained in \succ we have (i) $R \models (C \cdot \theta)$ or (ii) there exists a ground closure $(D \cdot \theta) \in N$ such that $(D \cdot \theta) \prec_R (C \cdot \theta)$ and $R \not\models (D \cdot \theta)$.

Definition 12. Let N be a set of ground closures. A ground inference ι with right or only premise $(C \cdot \theta)$ is called redundant w.r.t. N , if for every left-reduced ground rewrite system R contained in \succ we have (i) $R \models \text{concl}(\iota)$, or (ii) there exists a ground closure $(C' \cdot \theta) \in N$ such that $(C' \cdot \theta) \ll_R (C \cdot \theta)$ and $R \not\models (C' \cdot \theta)$, or (iii) ι is a SUPERPOSITION inference with left premise $(D' \vee t \approx t' \cdot \theta)$ where $t\theta \succ t'\theta$, and $(t\theta \rightarrow t'\theta) \notin R$, or (iv) ι is a SUPERPOSITION inference where the left premise is not the \succ_R -minimal premise.

Intuitively, a redundant closure cannot be a minimal counterexample, i.e., a minimal closure that is false in R . A redundant inference is either irrelevant for the completeness proof (cases (iii) and (iv)), or its conclusion (and thus its right or only premise) is true in R , provided that all closures that are \succ_R -smaller than the right or only premise are true in R (cases (i) and (ii)) – which means that the inference can be used to show that the right or only premise cannot be a minimal counterexample.

We denote the set of redundant closures w.r.t. N by $Red_C(N)$ and the set of redundant inferences by $Red_I(N)$.

Example 13. Let \succ be a KBO where all symbols have weight 1. Let $C = g(b) \not\approx c \vee f(c) \not\approx d$ and $C' = f(g(b)) \not\approx d$. Then the closure $(C \cdot \emptyset)$ is redundant w.r.t. $\{(C' \cdot \emptyset)\}$: Let R be a left-reduced ground rewrite system contained in \succ . Assume that C is false in R . Then $g(b)$ and c have the same R -normal form. Consequently, every redex or normal form in $\text{nm}_R(C' \cdot \emptyset)$ was already present in $\text{nm}_R(C \cdot \emptyset)$. Moreover, the labeled normal form $(c \downarrow_R, 2)$ that is present in $\text{nm}_R(C \cdot \emptyset)$ is missing in $\text{nm}_R(C' \cdot \emptyset)$. Therefore $(C \cdot \emptyset) \succ_R (C' \cdot \emptyset)$. Besides, if $(C \cdot \emptyset)$ is false in R , then $(C' \cdot \emptyset)$ is false as well.

Note that $C \prec_C C'$, therefore C is not classically redundant w.r.t. $\{C'\}$.

Lemma 14. (Red_I, Red_C) is a redundancy criterion in the sense of (Waldmann et al. [8]), that is, (1) if $N \models \perp$, then $N \setminus Red_C(N) \models \perp$; (2) if $N \subseteq N'$, then $Red_C(N) \subseteq Red_C(N')$ and $Red_I(N) \subseteq Red_I(N')$; (3) if $N' \subseteq Red_C(N)$, then $Red_C(N) \subseteq Red_C(N \setminus N')$ and $Red_I(N) \subseteq Red_I(N \setminus N')$; and (4) if ι is an inference with conclusion in N , then $\iota \in Red_I(N)$.

Proof. (1) Suppose that $N \setminus Red_C(N) \not\models \perp$. Then there exists a left-reduced ground rewrite system R contained in \succ such that $R \models N \setminus Red_C(N)$. We show that $R \models N$ (which implies $N \not\models \perp$). Assume that $R \not\models N$. Then there exists a closure $(C \cdot \theta) \in N \cap Red_C(N)$ such that $R \not\models (C \cdot \theta)$. By well-foundedness of \succ_R there exists a \succ_R -minimal closure $(C \cdot \theta)$ with this property. By definition of $Red_C(N)$, there must be a ground closure $(D \cdot \theta) \in N$ such that $(D \cdot \theta) \ll_R (C \cdot \theta)$ and $R \not\models (D \cdot \theta)$. By minimality of $(C \cdot \theta)$, we get $(D \cdot \theta) \in N \setminus Red_C(N)$, contradicting the initial assumption.

(2) Obvious.

(3) Let $N' \subseteq Red_C(N)$ and let $(C \cdot \theta) \in Red_C(N)$. We show that $(C \cdot \theta) \in Red_C(N \setminus N')$. Choose R arbitrarily. If $R \models (C \cdot \theta)$, we are done. Otherwise there exists a ground closure $(D \cdot \theta) \in N$ such that $(D \cdot \theta) \ll_R (C \cdot \theta)$ and $R \not\models (D \cdot \theta)$. By well-foundedness of \succ_R there exists a \succ_R -minimal closure $(D \cdot \theta)$ with this

property. If $(D \cdot \theta)$ were contained in N' and hence in $Red_C(N)$, there would exist a ground closure $(D' \cdot \theta) \in N$ such that $(D' \cdot \theta) \prec_R (D \cdot \theta)$ and $R \not\models (D' \cdot \theta)$, contradicting minimality. Therefore $(D \cdot \theta) \in N \setminus N'$ as required. The second part of (3) is proved analogously.

(4) Let ι be an inference with $\text{concl}(\iota) \in N$. Choose R arbitrarily. We have to show that ι satisfies part (i), (ii), (iii), or (iv) of Definition 12. Assume that (i), (iii), and (iv) do not hold. Then $R \not\models \text{concl}(\iota)$, and by Lemmas 9 and 10, $\text{concl}(\iota)$ is \succ_R -smaller than the right or only premise of ι , therefore part (ii) is satisfied if we take $\text{concl}(\iota)$ as $(C' \cdot \theta)$. \square

Constructing a Candidate Interpretation. In usual completeness proofs for superposition-like calculi, one constructs a candidate interpretation (a set of ground rewrite rules) for a saturated set of ground clauses by induction over the *clause ordering*. In our case, this is impossible since the limit *closure ordering* depends on the generated set of rewrite rules itself. We can still construct the candidate interpretation by induction over the *term ordering*, though: Instead of inspecting ground closures one by one as in the classical construction, we inspect all ground closures $(C \cdot \theta)$ for which $C\theta$ contains the maximal term s simultaneously, and if for at least one of them the usual conditions for productivity are satisfied, we choose the \succ_{R_s} -smallest one of these to extend R_s .

Let N be a set of ground closures. For every ground term s we define $R_s = \bigcup_{t \prec_s} E_t$. Furthermore we define $E_s = \{s \rightarrow s'\}$, if $(C \cdot \theta)$ is the \succ_{R_s} -smallest closure in N such that $C = C' \vee u \approx u'$, $s = u\theta$ is a strictly maximal term in $C\theta$, occurs only in a positive literal of $C\theta$, and is irreducible w.r.t. R_s , $s' = u'\theta$, $C\theta$ is false in R_s , and $s \succ s'$, provided that such a closure $(C \cdot \theta)$ exists. We say that $(C \cdot \theta)$ produces $s \rightarrow s'$. If no such closure exists, we define $E_s = \emptyset$. Finally, we define $R_* = \bigcup_t E_t$.

The following two lemmas are proved as usual:

Lemma 15. *Let s be a ground term, let $(C \cdot \theta)$ be a closure. If every term that occurs in negative literals of $C\theta$ is smaller than s and every term that occurs in positive literals of $C\theta$ is smaller than or equal to s , and if $R_s \models (C \cdot \theta)$, then $R_* \models (C \cdot \theta)$.*

Lemma 16. *If a closure $(C' \vee u \approx u' \cdot \theta)$ produces $u\theta \rightarrow u'\theta$, then $R_* \models (C' \vee u \approx u' \cdot \theta)$ and $R_* \not\models (C' \cdot \theta)$.*

Lemma 17. *Let $(C_1 \cdot \theta)$ and $(C_2 \cdot \theta)$ be two closures. If s is a strictly maximal term and occurs only positively in both $C_1\theta$ and $C_2\theta$, then $(C_1 \cdot \theta) \succ_{R_s} (C_2 \cdot \theta)$ if and only if $(C_1 \cdot \theta) \succ_{R_*} (C_2 \cdot \theta)$.*

Lemma 18. *Let $(D \cdot \theta) = (D' \vee t \approx t' \cdot \theta)$ and $(C \cdot \theta)$ be two closures in N . If $(D \cdot \theta)$ produces $t\theta \rightarrow t'\theta$ in R_* , and $t\theta$ occurs in $C\theta$ in a negative literal or below the top a term in a positive literal, then $(D \cdot \theta) \prec_{R_*} (C \cdot \theta)$ and $D\theta \prec_C C\theta$.*

Lemma 19. *Let $(D \cdot \theta) = (D' \vee t \approx t' \cdot \theta)$ and $(C \cdot \theta)$ be two closures in N . If $(D \cdot \theta)$ produces $t\theta \rightarrow t'\theta$ in R_* , $t\theta$ occurs in $C\theta$ at the top of the strictly maximal side of a positive maximal literal, and $R_* \not\models (C \cdot \theta)$, then $(D \cdot \theta) \prec_{R_*} (C \cdot \theta)$.*

We can now show that the Ground Closure Horn Superposition Calculus is refutationally complete:

Theorem 20. *Let N be a saturated set of ground closures that does not contain $(\perp \cdot \theta)$. Then $R_* \models N$.*

Proof. Suppose that $R_* \not\models N$. Let $(C \cdot \theta)$ be the \succ_{R_*} -smallest closure in N such that $R_* \not\models (C \cdot \theta)$.

Case 1: $C = C' \vee s \not\approx s'$ and $s\theta \not\approx s'\theta$ is maximal in $C\theta$. By assumption, $R_* \not\models s\theta \not\approx s'\theta$, hence $s\theta \downarrow_{R_*} = s'\theta \downarrow_{R_*}$.

Case 1.1: $s\theta = s'\theta$. Then there is an EQUALITY RESOLUTION inference from $(C \cdot \theta)$ with conclusion $(C'\sigma \cdot \theta)$, where $\theta \circ \sigma = \theta$. By saturation the inference is redundant, and by minimality of $(C \cdot \theta)$ w.r.t. \succ_{R_*} this implies $R_* \models (C'\sigma \cdot \theta)$. But then $R_* \models (C \cdot \theta)$, contradicting the assumption.

Case 1.2: $s\theta \neq s'\theta$. W.l.o.g. let $s\theta \succ s'\theta$. Then $s\theta$ must be reducible by a rule $t\theta \rightarrow t'\theta \in R_*$, which has been produced by a closure $(D \cdot \theta) = (D' \vee t \approx t' \cdot \theta)$ in N . By Lemma 18, $(D \cdot \theta) \prec_{R_*} (C \cdot \theta)$ and $D\theta \prec_C C\theta$. If $s\theta$ and $t\theta$ overlap at a non-variable position of s , there is a PARALLEL SUPERPOSITION I inference ι between $(D \cdot \theta)$ and $(C \cdot \theta)$; otherwise they overlap at or below a variable position of s and there is a PARALLEL SUPERPOSITION II inference ι with premises $(D \cdot \theta)$ and $(C \cdot \theta)$. By Lemma 16, $R_* \not\models (D' \cdot \theta)$. By saturation the inference is redundant, and by minimality of $(C \cdot \theta)$ w.r.t. \succ_{R_*} we know that $R_* \models \text{concl}(\iota)$. Since $R_* \not\models (D' \cdot \theta)$ this implies $R_* \models (C \cdot \theta)$, contradicting the assumption.

Case 2: $C\theta = C'\theta \vee s\theta \approx s'\theta$ and $s\theta \approx s'\theta$ is maximal in $C\theta$. By assumption, $R_* \not\models s\theta \approx s'\theta$, hence $s\theta \downarrow_{R_*} \neq s'\theta \downarrow_{R_*}$. W.l.o.g. let $s\theta \succ s'\theta$.

Case 2.1: $s\theta$ is reducible by a rule $t\theta \rightarrow t'\theta \in R_*$, which has been produced by a closure $(D \cdot \theta) = (D' \vee t \approx t' \cdot \theta)$ in N . By Lemmas 18 and 19, we obtain $(D \cdot \theta) \prec_{R_*} (C \cdot \theta)$, and, provided that $t\theta$ occurs in $s\theta$ below the top, also $D\theta \prec_C C\theta$. Therefore there is a PARALLEL SUPERPOSITION (I or II) inference ι with left premise $(D \cdot \theta)$ and right premise $(C \cdot \theta)$, and we can derive a contradiction analogously to Case 1.2.

Case 2.2: It remains to consider the case that $s\theta$ is irreducible by R_* . Then $s\theta$ is also irreducible by $R_{s\theta}$. Furthermore, by Lemma 17, $\succ_{R_{s\theta}}$ and \succ_{R_*} agree on all closures in which $s\theta$ is a strictly maximal term and occurs only positively. Therefore $(C \cdot \theta)$ satisfies all conditions for productivity, hence $R_* \models (C \cdot \theta)$, contradicting the assumption. \square

4.3 Lifting

It remains to lift the refutational completeness result for ground closure Horn superposition to the non-ground case.

If C is a general clause, we call every ground closure $(C \cdot \theta)$ a ground instance of C . If

$$\frac{C_n \dots C_1}{C_0}$$

is a general inference and

$$\frac{(C_n \cdot \theta) \dots (C_1 \cdot \theta)}{(C_0 \cdot \theta)}$$

is a ground inference, we call the latter a ground instance of the former. The function \mathcal{G} maps every general clause C and every general inference ι to the set of its ground instances. We extend \mathcal{G} to sets of clauses N or sets of inferences I by defining $\mathcal{G}(N) := \bigcup_{C \in N} \mathcal{G}(C)$ and $\mathcal{G}(I) := \bigcup_{\iota \in I} \mathcal{G}(\iota)$.

Lemma 21. *\mathcal{G} is a grounding function, that is, (1) $\mathcal{G}(\perp) = \{\perp\}$; (2) if $\perp \in \mathcal{G}(C)$ then $C = \perp$; and (3) for every inference ι , $\mathcal{G}(\iota) \subseteq \text{Red}_I(\mathcal{G}(\text{concl}(\iota)))$.*

The grounding function \mathcal{G} induces a lifted redundancy criterion $(\text{Red}_I^{\mathcal{G}}, \text{Red}_C^{\mathcal{G}})$ where $\iota \in \text{Red}_I^{\mathcal{G}}(N)$ if and only if $\mathcal{G}(\iota) \subseteq \text{Red}_I(\mathcal{G}(N))$ and $C \in \text{Red}_C^{\mathcal{G}}(N)$ if and only if $\mathcal{G}(C) \subseteq \text{Red}_C(\mathcal{G}(N))$.

Lemma 22. *Every ground inference from closures in $\mathcal{G}(N)$ is a ground instance of an inference from N or contained in $\text{Red}_I(\mathcal{G}(N))$.*

Proof. Let ι be a ground inference from closures in $\mathcal{G}(N)$. If ι is a PARALLEL SUPERPOSITION I or EQUALITY RESOLUTION inference, then it is a ground instance of a PARALLEL SUPERPOSITION or EQUALITY RESOLUTION with premises in N . It remains to consider PARALLEL SUPERPOSITION II inferences

$$\frac{(D' \vee t \approx t' \cdot \theta) \quad (C \cdot \theta)}{(D' \vee C \cdot \theta[x \mapsto u[t'\theta]])}$$

with $x\theta = u[t\theta]$. Let R be a left-reduced ground rewrite system contained in \succ . If $(t\theta \rightarrow t'\theta) \notin R$, then ι satisfies case (iii) of Definition 12. Otherwise $(C \cdot \theta[x \mapsto u[t'\theta]])$ is a ground instance of the clause $C \in N$ and \succ_R -smaller than the premise $(C \cdot \theta)$. If $R \models (C \cdot \theta[x \mapsto u[t'\theta]])$, then $R \models \text{concl}(\iota)$, so ι satisfies case (i) of Definition 12; otherwise it satisfies case (ii) of Definition 12. \square

Theorem 23. *The Horn Superposition Calculus (using PARALLEL SUPERPOSITION) together with the lifted redundancy criterion $(\text{Red}_I^{\mathcal{G}}, \text{Red}_C^{\mathcal{G}})$ is refutationally complete.*

Proof. This follows immediately from Lemma 22 and Theorem 32 from (Waldmann et al. [8]). \square

4.4 Deletion and Simplification

It turns out that DER, as well as most concrete deletion and simplification techniques that are implemented in state-of-the-art superposition provers are in fact covered by our abstract redundancy criterion. There are some unfortunate exceptions, however.

DER. DESTRUCTIVE EQUALITY RESOLUTION, that is, the replacement of a clause $x \not\approx t \vee C$ with $x \notin \text{vars}(t)$ by $C\{x \mapsto t\}$ is covered by the redundancy criterion. To see this, consider an arbitrary ground instance $(x \not\approx t \vee C \cdot \theta)$ of $x \not\approx t \vee C$. Let R be a left-reduced ground rewrite system contained in \succ . Assume that the instance is false in R . Then $x\theta$ and $t\theta$ have the same R -normal form. Consequently, any redex or normal form in $\text{nm}_R(C\{x \mapsto t\} \cdot \theta)$ was already present in $\text{nm}_R(x \not\approx t \vee C \cdot \theta)$ (possibly with a larger label, if x occurs only positively in C). Moreover, the labeled normal form $(x\theta \downarrow_R, 2)$ that is present in $\text{nm}_R(x \not\approx t \vee C \cdot \theta)$ is missing in $\text{nm}_R(C\{x \mapsto t\} \cdot \theta)$. Therefore $(x \not\approx t \vee C \cdot \theta) \not\approx_R (C\{x \mapsto t\} \cdot \theta)$. Besides, both closures have clearly the same truth value in R , that is, false.

Subsumption. Propositional subsumption, that is, the deletion of a clause $C \vee D$ with nonempty D in the presence of a clause C is covered by the redundancy criterion. This follows from the fact that every ground instance $((C \vee D) \cdot \theta)$ of the deleted clause is entailed by a smaller ground instance $(C \cdot \theta)$ of the subsuming clause. This extends to all simplifications that replace a clause by a subsuming clause in the presence of certain other clauses, for instance the replacement of a clause $t\sigma \approx t'\sigma \vee C$ by C in the presence of a clause $t \not\approx t'$, or the replacement of a clause $u[t\sigma] \not\approx u[t'\sigma] \vee C$ by C in the presence of a clause $t \approx t'$.

First-order subsumption, that is, the deletion of a clause $C\sigma \vee D$ in the presence of a clause C is not covered, however. This is due to the fact that \succ_R makes the instance $C\sigma$ smaller than C , rather than larger (see Lemma 7).

Tautology Deletion. The deletion of (semantic or syntactic) tautologies is obviously covered by the redundancy criterion.

Parallel Rewriting with Condition Literals. Parallel rewriting with condition literals, that is, the replacement of a clause $t \not\approx t' \vee C[t_1, \dots, t_k]_{p_1, \dots, p_k}$, where $t \succ t'$ and p_1, \dots, p_k are all the occurrences of t in C by $t \not\approx t' \vee C[t'_1, \dots, t'_k]_{p_1, \dots, p_k}$ is covered by the redundancy criterion. This can be shown analogously as for DER.

Demodulation. Parallel demodulation is the replacement of a clause $C[t\sigma, \dots, t\sigma]_{p_1, \dots, p_k}$ by $C[t'\sigma, \dots, t'\sigma]_{p_1, \dots, p_k}$ in the presence of another clause $t \approx t'$ where $t\sigma \succ t'\sigma$. In general, this is *not* covered by our redundancy criterion. For instance, if \succ is a KBO where all symbols have weight 1 and if $R = \{f(b) \rightarrow b, g(g(b)) \rightarrow b\}$, then replacing $f(f(f(b)))$ by $g(g(b))$ in some clause $f(f(f(b))) \not\approx c$ yields a clause with a larger R -normalization multiset, since the labeled redexes $\{(f(b), 2), (f(b), 2), (f(b), 2)\}$ are replaced by $\{(g(g(b)), 2)\}$ and $g(g(b)) \succ f(b)$.

A special case is supported, though: If $t'\sigma$ is a proper subterm of $t\sigma$, then the R -normalization multiset either remains the same or becomes smaller, since every redex in the normalization of $t'\sigma$ occurs also in the normalization of $t\sigma$.

5 Completeness, Part II: The Non-horn Case

In the non-Horn case, the construction that we have seen in the previous section fails for (PARALLEL) SUPERPOSITION inferences at the top of positive literals. Take an LPO with precedence $f \succ c_6 \succ c_5 \succ c_4 \succ c_3 \succ c_2 \succ c_1 \succ b$ and consider the ground closures $(f(x_1) \approx c_1 \vee f(x_2) \approx c_2 \vee f(x_3) \approx c_3 \cdot \theta)$ and $(f(x_4) \approx c_4 \vee f(x_5) \approx c_5 \vee f(x_6) \approx c_6 \cdot \theta)$, where θ maps all variables to the same constant b . Assume that the first closure produces the rewrite rule $(f(b) \approx c_3) \in R_*$. The R_* -normalization multisets of both closures are dominated by three occurrences of the labeled redex $(f(b), 0)$. However, a SUPERPOSITION inference between the closures yields $(f(x_1) \approx c_1 \vee f(x_2) \approx c_2 \vee f(x_4) \approx c_4 \vee f(x_5) \approx c_5 \vee c_3 \approx c_6 \cdot \theta)$, whose R_* -normalization multiset contains four occurrences of the labeled redex $(f(b), 0)$, hence the conclusion of the inference is larger than both premises. If we want to change this, we must ensure that the weight of positive literals depends primarily on their larger sides, and if the larger sides are equal, on their smaller sides. That means that in the non-Horn case, the clause ordering must treat positive literals as the traditional clause ordering \succ_C . But that has two important consequences: First, DER may no longer be used to eliminate variables that occur also in positive literals (since DER might now increase the weight of these literals). On the other hand, unrestricted demodulation becomes possible for positive literals.

We sketch the key differences between the non-Horn and the Horn case; for the details, we refer to the technical report [7].

We define the subterm set $ss^-(C)$ and the topterm set $ts^-(C)$ of a clause C as in the Horn case:

$$\begin{aligned} ss^-(C) &= \{t \mid C = C' \vee s[t]_p \not\approx s'\} \\ ts^-(C) &= \{t \mid C = C' \vee t \not\approx t'\} \end{aligned}$$

We do not need labels anymore. Instead, for every redex or normal form u that appears in negative literals we include the two-element multiset $\{u, u\}$ in the R -normalization multiset to ensure that a redex or normal form u in negative literals has a larger weight than a positive literal $u \approx v$ with $u \succ v$. We define the R -redex multiset $rm_R(t)$ of a ground term t by

$$\begin{aligned} rm_R(t) &= \emptyset \text{ if } t \text{ is } R\text{-irreducible;} \\ rm_R(t) &= \{\{u, u\}\} \cup rm_R(t') \text{ if } t \rightarrow_R t' \text{ using the rule } u \rightarrow v \in R. \end{aligned}$$

The R -normalization multiset $nm_R(C \cdot \theta)$ of a ground closure $(C \cdot \theta)$ is

$$\begin{aligned} nm_R(C \cdot \theta) &= \bigcup_{f(t_1, \dots, t_n) \in ss^-(C)} rm_R(f(t_1 \theta \downarrow_R, \dots, t_n \theta \downarrow_R)) \\ &\quad \cup \bigcup_{x \in ss^-(C)} rm_R(x\theta) \\ &\quad \cup \bigcup_{t \in ts^-(C)} \{\{t\theta \downarrow_R, t\theta \downarrow_R\}\} \\ &\quad \cup \bigcup_{(s \approx s') \in C} \{\{s\theta, s'\theta\}\} \end{aligned}$$

Once more, the R -normalization closure ordering \succ_R compares ground closures $(C \cdot \theta_1)$ and $(D \cdot \theta_2)$ using a lexicographic combination of three orderings:

- first, the twofold multiset extension $(\succ_{\text{mul}})_{\text{mul}}$ of the reduction ordering \succ applied to the multisets $\text{nm}_R(C \cdot \theta_1)$ and $\text{nm}_R(D \cdot \theta_2)$,
- second, the traditional clause ordering \succ_c applied to $C\theta_1$ and $D\theta_2$,
- and third, the same closure ordering \succ_{clo} as in the Horn case.

With this ordering, we can again prove Lemmas 9 and 10 and their analogue for EQUALITY FACTORING, which implies Lemma 14. In the construction of a candidate interpretation, we define again $R_s = \bigcup_{t \prec_s} E_t$ for every ground term s and $R_* = \bigcup_t E_t$. We define $E_s = \{s \rightarrow s'\}$, if $(C \cdot \theta)$ is the \succ_{R_s} -smallest closure in N such that $C = C' \vee u \approx u'$, $u\theta \approx u'\theta$ is a strictly maximal literal in $C\theta$, $s = u\theta$, $s' = u'\theta$, $s \succ s'$, s is irreducible w.r.t. R_s , $C\theta$ is false in R_s , and $C'\theta$ is false in $R_s \cup \{s \rightarrow s'\}$, provided that such a closure $(C \cdot \theta)$ exists. If no such closure exists, we define $E_s = \emptyset$.

We can then reprove Theorem 20 for the non-Horn case. The only difference in the proof is one additional subcase before Case 2.1: If $s\theta \approx s'\theta$ is maximal, but not strictly maximal in $C\theta$, or if $C'\theta$ is true in $R_{s\theta} \cup \{s\theta \rightarrow s'\theta\}$, then there is an EQUALITY FACTORING inference with the premise $(C \cdot \theta)$. This inference must be redundant, which yields again a contradiction.

The lifting to non-ground clauses works as in Sect. 4.3.

6 Discussion

We have demonstrated that the naive addition of DESTRUCTIVE EQUALITY RESOLUTION (DER) to the standard abstract redundancy concept destroys the refutational completeness of the calculus, but that there exist restricted variants of the Superposition Calculus that are refutationally complete even with DER (restricted to negative literals in the non-Horn case). The key tool for the completeness proofs is a closure ordering that is structurally very different from the classical ones – it is not a multiset extension of some literal ordering – but that still has the property that the redundancy criterion induced by it is compatible with the Superposition Calculus.

Of course there is a big gap between the negative result and the positive results. The new redundancy criterion justifies DER as well as most deletion and simplification techniques found in realistic saturation provers, but only propositional subsumption and only a very restricted variant of demodulation. The question whether the Superposition Calculus is refutationally complete together with a redundancy criterion that justifies both DER (in full generality even in the non-Horn case) and *all* deletion and simplification techniques found in realistic saturation provers (including unrestricted demodulation and first-order subsumption) is still open. Our work is an intermediate step towards a solution to this problem. There may exist a more refined closure ordering that allows us to prove the completeness of such a calculus. On the other hand, if the combination is really incomplete, a counterexample must make use of those operations that our proof fails to handle, that is, DER in positive literals in non-Horn problems, first-order subsumption, or demodulation with unit equations that are contained in the usual term ordering \succ but yield closures that are larger w.r.t. \succ_{R_*} .

Acknowledgement. I thank the anonymous IJCAR reviewers for their helpful comments and Stephan Schulz for drawing my attention to the problem at the CASC dinner in 2013.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge, UK (1998)
2. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.* **4**(3), 217–247 (1994)
3. Bachmair, L., Ganzinger, H., Lynch, C., Snyder, W.: Basic paramodulation. *Inf. Comput.* **121**(2), 172–192 (1995)
4. Duarte, A., Korovin, K.: Ground joinability and connectedness in the superposition calculus. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) *Automated Reasoning, IJCAR 2022*, pp. 169–187. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_11
5. Nieuwenhuis, R., Rubio, A.: Theorem proving with ordering and equality constrained clauses. *J. Symb. Comput.* **19**(4), 321–351 (1995)
6. Schulz, S.: E—a brainiac theorem prover. *AI Commun.* **15**(2–3), 111–126 (2002)
7. Waldmann, U.: On the (in-)completeness of destructive equality resolution in the superposition calculus. Tech. rep., [arXiv:2405.03367](https://arxiv.org/abs/2405.03367)
8. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. *J. Autom. Reason.* **66**, 499–539 (2022). <https://doi.org/10.1007/s10817-022-09621-7>

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



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



SAT, SMT and Quantifier Elimination



Model Completeness for Rational Trees

Silvio Ghilardi^(✉)  and Lia M. Poidomani 

Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy

silvio.ghilardi@unimi.it

<http://users.mat.unimi.it/users/ghilardi/>

Abstract. We analyze the theory of rational trees with finitely many constructors, infinitely many atoms and an atomicity predicate. We design a new decision procedure, proving in addition that this theory is model-complete. We also show that the enrichment of the language with selectors and simultaneous parametric fixpoints enjoys quantifier elimination.

Keywords: Infinite Trees · Model Completeness · Decision Procedures

1 Introduction

The theory of finite and infinite trees deserves special attention in computer science applications for its capability of representing both algebraic and co-algebraic datatypes, including for instance (acyclic or even cyclic) lists, streams, etc. The theory has been largely investigated by the logic programming community since its early days [3, 8] and various results have been proved about it, including decidability in the full elementary language [12], albeit within very high (actually non elementary) complexity bounds [21]. The automated reasoning community gave some contributions too, focusing especially on the more restricted constraint solving fragment [20].

Deciding satisfiability of the elementary theory of finite and infinite trees cannot be obtained via quantifier elimination, as pointed out in the comprehensive and detailed paper [4], which improves and extends classical results from [12]. In fact, the normalizations performed by the algorithms proposed in these papers prove that every formula is equivalent to a Boolean combination of special kinds of primitive formulae (thus giving, in terms of prenex forms, a reduction to a $\Delta_2^0 = \exists^*\forall^* \cap \forall^*\exists^*$ -class), but not to a quantifier-free formula. The impossibility of full quantifier-elimination can indeed be shown by easy counterexamples. However, quantifier elimination is a nice and desirable property, for instance it facilitates the computation of interpolants [10], whose employment in verification applications is widely recognized [15].

From the point of view of ‘abstract logical nonsense’, quantifier elimination can be in principle always obtained, by the trivial enlargement of the language naming every formula by a fresh atomic predicate. More concretely, it often happens that manageable enrichments of the language are sufficient to achieve quantifier elimination: a classical example in this sense is the theory of real closed fields which becomes quantifier-eliminable after adding to the language the ordering predicate (this is easily seen to be

The first author is supported by INdAM’s GNSAGA group.

© The Author(s) 2024

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 265–283, 2024.

https://doi.org/10.1007/978-3-031-63498-7_16

definable in the theory, because ‘being positive’ turns out to be equivalent to ‘being a square’). A more complicated, but still manageable example is linear integer arithmetic, where in order to obtain quantifier elimination it is sufficient to enrich the language with the infinitely many definable binary predicates expressing ‘equivalence modulo n ’.

We take into consideration the variant of the theory of trees whose signature has *finitely many constructors, infinitely many constants symbols and an atomicity predicate*. We feel that this variant of the theory is rather natural to consider, especially in verification applications, where constants can represent infinite data (maybe constrained by richer theories describing their internal structure). The problem we are addressing in this paper is: what is the price to pay (in terms of language enrichments) to achieve quantifier elimination for the above theory? In [14] quantifier elimination is obtained by introducing *ad hoc* syntactic entities (called ‘terms with pointers’) and applying to them an extension of the classical Mal’cev quantifier elimination algorithm for finite trees [13]. In this paper we stay inside the native first order language and we prove that our theory is *model-complete* by introducing a novel technique based on *definability analysis*; as a by-product, we achieve full quantifier elimination by enriching the language with extra operation symbols for *selectors* and *simultaneous parametric fixpoints*.

The paper is structured as follows: Sects. 2, 3 introduce preliminary definitions, our variant of the theory of trees and establish some basic properties; Sect. 4 gives a new decision procedure for constraint solving problems using *graph representations, bisimulations and congruence closure*. Section 5 introduces our main technical tool, namely *definability in existential formulae*; Sect. 6 proves first model-completeness and then shows how to enrich the language to achieve full quantifier elimination. Section 7 concludes and discusses further improvements. The paper is meant to be self-contained; we moved to the online available extended version

http://users.mat.unimi.it/users/ghilardi/allegati/GP_IJCAR24.pdf

the proofs of few results which are either well-known or rather straightforward. Such extended version contains also a thorough comparison with the approach of some relevant papers from the literature like [4] and [14].

2 Preliminaries

We adopt the usual first-order syntactic notions of signature, term, atom, literal, (ground) formula, and so on; our signatures always include equality. We compactly represent a tuple of distinct variables as \underline{x} . The notation $t(\underline{x}), \phi(\underline{x})$ means that the term t , the formula ϕ has free variables included in the tuple \underline{x} . Since our *tuples of variables* are assumed to be formed by *distinct* elements, we underline that when we write e.g. $\phi(\underline{x}, \underline{y})$, we mean that the tuples $\underline{x}, \underline{y}$ are made of distinct variables and are also disjoint from each other. Notations like $\phi(\underline{t}/\underline{x})$ are used to denote substitutions.

A formula is said to be *universal* (resp., *existential*) if it has the form $\forall \underline{x} \phi(\underline{x}, \underline{y})$ (resp., $\exists \underline{x} \phi(\underline{x}, \underline{y})$), where ϕ is quantifier-free. Formulae with no free variables are called *sentences*. A *constraint* is a conjunction of literals; a *primitive* formula is obtained from a constraint by prefixing it a finite string of existential quantifiers.

From the semantic side, given a signature Σ , we use the standard notion of a Σ -structure \mathcal{M} and of truth of a formula in a Σ -structure under a free variables assignment. A Σ -theory \mathcal{T} is a set of Σ -sentences; a *model* of \mathcal{T} is a Σ -structure \mathcal{M} where all sentences in \mathcal{T} are true. We use the standard notation $\mathcal{T} \models \phi$ to say that ϕ is \mathcal{T} -valid, i.e. true in all models of \mathcal{T} for every assignment to the variables occurring free in ϕ . We say that two formulae ϕ and ψ are \mathcal{T} -equivalent iff $\phi \leftrightarrow \psi$ is \mathcal{T} -valid. A theory \mathcal{T} is *complete* iff for every sentence ϕ , either ϕ or $\neg\phi$ is \mathcal{T} -valid. Complete theories can be obtained from any Σ -structure \mathcal{M} by taking the set of the Σ -sentences that are true in \mathcal{M} (such a theory is denoted by $Th(\mathcal{M})$).

We say that ϕ is \mathcal{T} -satisfiable iff there is a model \mathcal{M} of \mathcal{T} and an assignment to the variables occurring free in ϕ making ϕ true in \mathcal{M} . The *elementary* (resp. *constraint*) *satisfiability problem* for \mathcal{T} is the following: we are given a formula (resp. a constraint) $\phi(\underline{x})$ and we are asked whether there exist a model \mathcal{M} of \mathcal{T} and an assignment α to the free variables \underline{x} such that $\mathcal{M}, \alpha \models \phi(\underline{x})$ holds, i.e. such that ϕ is true in \mathcal{M} under such assignment.

A theory \mathcal{T} has *quantifier elimination* iff for every formula $\phi(\underline{x})$ in the signature of \mathcal{T} there is a quantifier-free formula $\phi'(\underline{x})$ such that $\mathcal{T} \models \phi(\underline{x}) \leftrightarrow \phi'(\underline{x})$. We shall be interested also into a condition that is weaker than quantifier elimination, namely model-completeness. Such a notion is usually defined semantically (as the fact that an embedding between two models of the theory is elementary), however there is a well-known equivalent syntactic definition that we are going to extensively use (see [2], Thm. 3.5.1). This equivalent definition is supplied by the following:

Definition 1. *A theory \mathcal{T} is said to be model-complete iff every existential formula $\exists \underline{x} \phi(\underline{x}, \underline{y})$ in the signature of \mathcal{T} is \mathcal{T} -equivalent to a universal formula $\forall \underline{x}' \phi'(\underline{x}', \underline{y})$.*

Notice that, keeping in mind prenex normal forms and the interdefinability of universal and existential quantifiers, it turns out that in a model complete theory *every formula whatsoever* is \mathcal{T} -equivalent to *both* a universal and an existential formula; consequently, *model-completeness reduces the elementary satisfiability problem to the constraint satisfiability problem.*

3 Σ -Trees

In the whole paper we fix a signature (let us call it Σ) containing: (i) *infinitely many individual constants*, (ii) *finitely many function symbols* h_1, \dots, h_N of respective arities $ar_1, \dots, ar_N \geq 1$ and (iii) *a unary predicate* At . Symbols h_1, \dots, h_N are called Σ -constructors or just *constructors*. We use letters f, g, \dots for constructors or constants of Σ . We now define Σ -labelled trees.

A *tree* T is a set of finite lists of positive natural numbers satisfying the following three conditions; (1) the empty list ε belongs to T ; (2) T is prefix-closed, i.e. $\sigma * \tau \in T$ implies $\sigma \in T$ (here $*$ is list concatenation); (3) if for some positive numbers n, m we have $n < m$ and $\sigma * m \in T$, then $\sigma * n \in T$.

The elements of a tree T are called *nodes* and the node ε is called the *root* of T ; given a node $\sigma \in T$, the set $T_\sigma = \{\tau \mid \sigma * \tau \in T\}$ is a tree itself, called the σ -subtree of T . A *subtree* of T is a σ -subtree of T for some $\sigma \in T$.

A Σ -labelled tree (or just a Σ -tree) is a pair (T, Λ) such that T is a tree and Λ is a map $\Lambda : T \rightarrow \Sigma$ that associates with every node σ of T a constructor or a constant in such a way that if $\Lambda(\sigma)$ has arity n , then σ has precisely n -successors. The latter means that for every $i \geq 1$, we have $\sigma * i \in T$ iff $i \leq n$ (as a special case, $\sigma * i$ never belongs to T in case $\Lambda(\sigma)$ is a constant).

Given a node σ in a Σ -tree (T, Λ) , the σ -subtree T_σ of T can be endowed with a Σ -tree structure $(T_\sigma, \Lambda_\sigma)$ by putting $\Lambda_\sigma(\tau) = \Lambda(\sigma * \tau)$ for every $\tau \in T_\sigma$. The Σ -subtrees of (T, Λ) are the Σ -trees of the form $(T_\sigma, \Lambda_\sigma)$, varying σ among the nodes of T . If $\Lambda(\sigma)$ is a constructor whose arity is n , we call T -successors of σ the Σ -trees $(T_{\sigma*1}, \Lambda_{\sigma*1}), \dots, (T_{\sigma*n}, \Lambda_{\sigma*n})$; if $\Lambda(\sigma)$ is a constant, σ does not have T -successors and is said to be a leaf of T and of (T, Λ) .

Remark 1. If (T, Λ) is a Σ -tree, we use the notation $(\sigma, f) \in (T, \Lambda)$ to mean that $\sigma \in T$ and $\Lambda(\sigma) = f$. The above notation can be used in order to define a Σ -tree (T, Λ) by specifying by induction on the length of σ whether $(\sigma, f) \in (T, \Lambda)$ holds or not, for every σ and for every (constructor or constant) f .

A Σ -tree (T, Λ) is finite iff the set of nodes of T is finite, it is said to be infinite otherwise. (T, Λ) is rational iff the set of Σ -subtrees of (T, Λ) is finite; notice that a rational tree can be infinite. Now comes the important definition: the sets of Σ -trees, of finite Σ -trees and of rational Σ -trees give rise to Σ -structures in the following way:

- the interpretation of a constant $a \in \Sigma$ is the singleton Σ -tree $\{\varepsilon\}$ labelled with a ;
- the interpretation of the predicate At consists of the singleton Σ -trees;
- the interpretation of a constructor $h_i \in \Sigma$ of arity n maps the labelled trees $(T_1, \Lambda_1), \dots, (T_n, \Lambda_n)$ to the labelled tree (T, Λ) where

$$T = \{\varepsilon\} \cup \bigcup_{j=1}^n \{j * \sigma \mid \sigma \in T_j\}$$

and Λ is so defined

$$\Lambda(\varepsilon) = h_i, \quad \Lambda(j * \sigma) = \Lambda_j(\sigma)$$

(thus (T, Λ) is the Σ -tree whose root is labelled h_i and such that the T -successors of the root are $(T_1, \Lambda_1), \dots, (T_n, \Lambda_n)$).

It is well-known [12, 14] that the Σ -structure of all Σ -trees and of all rational Σ -trees are elementarily equivalent (i.e. the same Σ -sentences are true in them). So it makes no difference to work on rational trees or on all Σ -trees; in this paper we made the choice to work on the structure of rational Σ -trees. We call \mathcal{R} this structure. The related complete theory $Th(\mathcal{R})$ will be also called \mathcal{R} (so, for simplicity, we use the same letter \mathcal{R} for the set of rational Σ -trees, the related Σ -structure and the associated complete theory).

There are some important formulae valid in \mathcal{R} that we want to list below. In order to have a compact and intuitive notation, we need some abbreviations. If \underline{x} is the tuple x_1, \dots, x_n , let us use $\forall \underline{x} \phi$ for $\forall x_1 \dots \forall x_n \phi$ - a similar convention is used for $\exists \underline{x} \phi$. If $\underline{t} = t_1, \dots, t_n$ is a tuple of terms of the same length as \underline{x} , then $\underline{x} = \underline{t}$ stands for $\bigwedge_{i=1}^n x_i = t_i$; moreover $\exists! \underline{x} \phi(\underline{x}, \underline{y})$ is used for $\exists \underline{x} \phi(\underline{x}, \underline{y}) \wedge \forall \underline{x} \forall \underline{x}' (\phi(\underline{x}, \underline{y}) \wedge \phi(\underline{x}', \underline{y}) \rightarrow \underline{x} = \underline{x}')$.

It is useful to have abbreviations for formulas expressing that “ x is rooted by h_i ”, these are

$$R_{h_i}(x) : \equiv \exists x_1 \cdots \exists x_{ar_i}(x = h_i(x_1, \dots, x_{ar_i})) \quad (1)$$

for every $i = 1, \dots, N$ (recall that h_1, \dots, h_N are our constructors). The claims of the following Proposition are either immediate or well-known [4]:

Proposition 1. *The following sentences are \mathcal{R} -valid:*

- (i) $\forall \underline{x}(h_i(\underline{x}) = h_i(\underline{x}') \rightarrow \underline{x} = \underline{x}')$ (for $i = 1, \dots, N$);
- (ii) $\forall x \neg(R_{h_i}(x) \wedge R_{h_j}(x))$ (for $i \neq j$ and $i, j = 1, \dots, N$);
- (iii) $\forall x \neg(R_{h_i}(x) \wedge At(x))$ (for $i = 1, \dots, N$);
- (iv) $\forall x(At(x) \vee \bigvee_{i=1}^N R_{h_i}(x))$;
- (v) $At(a) \wedge a \neq b$ (for distinct constants $a, b \in \Sigma$);
- (vi) $\forall \underline{z} \exists! \underline{x}(\underline{x} = \underline{t}(\underline{x}, \underline{z}))$, where $\underline{t}(\underline{x}, \underline{z})$ are proper flat terms.

Above, a *flat* term is a constant, a variable, or a constructor applied to variables; a flat term is *proper* iff it is not a variable.

We make a comment on formula (vi) above. To correctly interpret it, recall that $\underline{x}, \underline{z}$ must be distinct tuples of variables including all variables occurring in \underline{t} according to our conventions. Formula (vi) expresses the *existence and uniqueness of simultaneous parametric fixpoints*: the fixpoints are simultaneous because \underline{x} is a tuple of variables (it is not a single variable) and the \underline{z} are the parameters on which the fixpoints depend.

We underline the following important fact, which is a consequence of the above Proposition: the formula $R_{h_i}(x)$ is *existential* according to its definition (1); however, because of Proposition 1(ii)–(iv), it is also logically equivalent to a *universal* formula, namely

$$\neg At(x) \wedge \bigwedge_{j \neq i} \neg R_{h_j}(x) \quad (2)$$

(this is typical of model-complete theories, see Definition 1).

A *flat literal* is a literal of one of the following forms

$$x = t, \quad x \neq y, \quad At(x), \quad \neg At(x)$$

where x, y are variables and t is a flat term; a flat literal is *proper* iff it is of the kind $At(x), x \neq y, x = t$, where x, y are variables and the term t is flat and proper. A proper flat literal of the kind $At(x)$ is said to be an *atomicity* proper flat literal, a proper flat literal of the kind $x \neq y$ is said to be a *disequality* proper flat literal, a proper flat literal of the kind $x = t$ is said to be an *equality* proper flat literal with *head variable* x .

Definition 2. *A constraint $C(\underline{x})$ is in solved form iff it is a conjunction of atomicity proper flat literals, disequality proper flat literals and equality proper flat literals whose head variables are pairwise different. We also require that if a constraint C in solved form contains an atomicity proper flat literal of the kind $At(v)$, then the variable v is not a head variable of any equality proper flat literal $v = t$ of C , unless t is a constant.*

We have an analogous notion for primitive formulae. A conjunction of variable equalities $E(\underline{x}', \underline{x})$ of the kind $x' = x$ for $x \in \underline{x}$ and $x' \in \underline{x}'$ is an *equivalence diagram* (*e-diagram* for short) iff for every $x' \in \underline{x}'$ there is exactly one equality $x' = x$ with $x \in \underline{x}$ that is a conjunct of E (that is, E is a ‘logical representation’ of an equivalence relation over $\underline{x} \cup \underline{x}'$, having the \underline{x} as representative elements for the equivalence classes).

The role of the e-diagrams in Definition 3 and in the algorithm of Proposition 2 below is subsequent to a choice of representative variables for equivalence classes: once a choice is done, e-diagrams neutralize unquantified non-representative variables (these variables cannot be eliminated), by moving them outside the scope of the existential quantifiers (in such position they cannot play any role in future manipulations).

Definition 3. *A primitive formula is in solved form iff it can be written (up to logical equivalence) in the form*

$$E(\underline{x}', \underline{x}) \wedge \exists \underline{y} C(\underline{x}, \underline{y}) \tag{3}$$

where $C(\underline{x}, \underline{y})$ is a constraint in solved form and $E(\underline{x}, \underline{x}')$ is an e-diagram.

A primitive formula in solved form (3) has the following important property: any assignment to the free variables \underline{x} satisfying C can be extended to an assignment to \underline{x}' satisfying $E(\underline{x}', \underline{x}) \wedge \exists \underline{y} C(\underline{x}, \underline{y})$. The following Proposition shows that the satisfiability of existential formulae can be reduced to satisfiability of primitive formulae in solved form:

Proposition 2. *Every existential formula is \mathcal{R} -equivalent to a disjunction of primitive formulae in solved form.*

Proof. We first flatten all terms occurring in our formula using fresh existentially quantified variables and the logical equivalence $\phi(x/t) \leftrightarrow \exists x(x = t \wedge \phi)$ (here x is supposed not to occur in t). Then we remove negative atom statements by the \mathcal{R} -equivalence $\neg A(t) \leftrightarrow \bigvee_{i=1}^N R_{h_i}(x)$. Applying DNF conversion and distributing the existential quantifier over disjunctions, we obtain a disjunction of primitive formulae whose matrices are conjunctions of flat literals.

Now we exhaustively apply to each disjunct of the form $\exists \underline{y} \phi(\underline{x}, \underline{y})$ the rewrite rules below (disjuncts containing an inconsistency $v \neq v$ are in the end removed). The rules modify the whole disjunct or some conjuncts of its as indicated. When specifying the rules, we use letters $x_i \in \underline{x}$ for the free variables \underline{x} of $\exists \underline{y} \phi(\underline{x}, \underline{y})$, letter $y \in \underline{y}$ for a quantified variable of $\exists \underline{y} \phi(\underline{x}, \underline{y})$ and letters $v, v_i, w_j \in \underline{x} \cup \underline{y}$ for any variable occurring in $\exists \underline{y} \phi(\underline{x}, \underline{y})$; we also fix a total order \prec on the free variables \underline{x} .

- (0) $v = v \wedge \psi \implies \psi$;
- (1) $At(v) \wedge v = t \implies v \neq v$ (if t is a non-variable, non-constant term);
- (2) $v = t \wedge v = t' \implies v \neq v$ (if t, t' are non-variable terms rooted with different constructors or constants);
- (3) $v = f(v_1, \dots, v_n) \wedge v = f(w_1, \dots, w_n) \implies v = f(w_1, \dots, w_n) \wedge \bigwedge_{i=1}^n v_i = w_i$;
- (4) $\exists \underline{y}(x_1 = x_2 \wedge \psi) \implies x_1 = x_2 \wedge \exists \underline{y}(\psi(x_1/x_2))$ (if $x_1 \prec x_2$; in case $x_2 \prec x_1$, the rule is applied by inverting the roles of x_1 and x_2);
- (5) $\exists \underline{y}(y = v \wedge \psi) \implies \psi(v/y)$ (where v is a variable different from y).

When applying Rule (4) and moving the equality $x_1 = x_2$ outside the existential quantifiers $\exists y$, we simultaneously replace any free variable equality of the kind $x' = x_2$ by $x' = x_1$ (this ensures that free variable equalities outside the existential quantifiers form an e-diagram). The above rules are justified either as logical equivalences or by Proposition 1. Rules (2)–(3) adjust equality proper flat literals whose head variables are the same, whereas Rules (4)–(5) eliminate variable equalities inside the scope of the existential quantifiers. For termination, we consider pairs of natural numbers (k_1, k_2) , where k_1 is the number of occurrences of constructors symbols and k_2 is the number of literals inside the existential quantifiers. All rules decrease k_2 (keeping k_1 unchanged), except Rule (3) which decreases k_1 .¹ \dashv

Remark 2. From the point of view of complexity, it is clear that DNF conversion produces an exponential blow-up; however Rules (0)–(5) can be exhaustively applied in polynomial (actually quadratic) time.

4 Σ -Graphs and Bisimulations

In this section, we introduce a procedure to test \mathcal{R} -satisfiability of constraints in solved form. The procedure is based on Σ -graphs (we shall essentially use Σ -graphs as alternative representations of Σ -trees):

Definition 4. A Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ is a set G endowed with an infinite family of unary partial functions R^i ($i \geq 1$) and with a labeling function $\lambda : G \rightarrow \Sigma$ satisfying the following condition: for every $i \geq 1$, the domain of the partial function R^i is the set of $g \in G$ such that the arity of $\lambda(g)$ is larger than or equal to i .

Notice that the domains of the R^i exclude the nodes of a Σ -graph labeled by a constant symbol. We use the notation $gR^i g'$ to say that $g \in \text{dom}(R^i)$ and $R^i(g) = g'$. We adapt to our context the classical notion of bisimulation:

Definition 5. A bisimulation between Σ -graphs $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ and $\mathcal{G}' = (G', \{R'_i\}_{i \geq 1}, \lambda')$ is a relation $\rho \subseteq G \times G'$ satisfying the following conditions:

- (i) for all $g \in G, g' \in G'$, if $\rho(g, g')$ then $\lambda(g) = \lambda(g')$;
- (ii) for all $g_1 \in G, g'_1, g'_2 \in G'$ and $i \geq 1$, if $\rho(g_1, g'_1)$ and $g'_1 R'_i g'_2$ then there exists $g_2 \in G$ such that $g_1 R^i g_2$ and $\rho(g_2, g'_2)$;
- (iii) for all $g_1, g_2 \in G, g'_1 \in G'$ and $i \geq 1$, if $\rho(g_1, g'_1)$ and $g_1 R^i g_2$ then there exists $g'_2 \in G'$ such that $g'_1 R'_i g'_2$ and $\rho(g_2, g'_2)$.

A bisimulation relation which is a total function $\rho : G \rightarrow G'$ is said to be a p-morphism of \mathcal{G} into \mathcal{G}' ; if ρ is also surjective, we say that \mathcal{G}' is a quotient of \mathcal{G} .

Since our R^i are partial functions and (i) holds, conditions (ii) and (iii) in the definition of a bisimulation are trivially seen to be equivalent to each other (so one of them is redundant).

¹ It is essential that Rule (3) is applied to flat constraints, otherwise it might cause non termination if naively formulated [16].

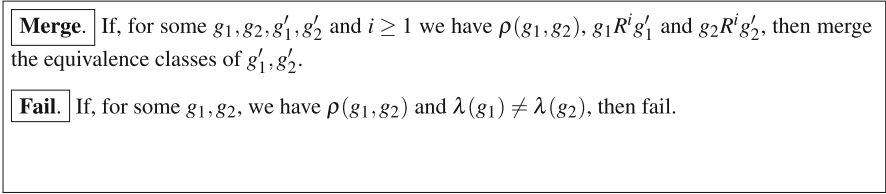


Fig. 1. Rules of Pseudo Congruence Closure Algorithm

Bisimulations are closed under unions, so that there is the *biggest bisimulation* among two given Σ -graphs \mathcal{G} and \mathcal{G}' . We write $g \sim_b g'$ to mean that there exists a bisimulation between the nodes g and g' of \mathcal{G} and \mathcal{G}' (equivalently, that we have $\rho(g, g')$ for the biggest bisimulation among \mathcal{G} and \mathcal{G}'). Also, since the identity relation is a bisimulation, the converse of a bisimulation is a bisimulation and the composition of bisimulations is a bisimulation, the relation \sim_b restricted to the nodes of the same Σ -graph \mathcal{G} turns out to be an equivalence relation. Bisimulation relations between a Σ -graph and itself which are equivalence relations (called *bisimulation equivalences*) produce quotients, in the sense explained by the following Proposition:

Proposition 3. *Suppose that ρ is a bisimulation equivalence on a Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$. Then there are a Σ -graph \mathcal{G}' and a quotient $q : \mathcal{G} \rightarrow \mathcal{G}'$ such that we have $\rho(g, g')$ iff $q(g) = q(g')$ for all g, g' in \mathcal{G} .*

Proof. We let G' be the quotient set of G under the equivalence relation ρ and q be the map associating with g its equivalence class $[g]$. We then put

$$\lambda'([g]) = \lambda(g), \quad [g]R'_i[g'] \text{ iff } \exists g'' \text{ s.t. } \rho(g', g'') \text{ and } gR^i g'' \text{ (for all } i \geq 0 \text{)} .$$

In this way $\mathcal{G}' = (G', \{R'_i\}_{i \geq 1}, \lambda')$ is a Σ -graph and q is the desired quotient. ◁

Given a *finite* Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ and a relation $\rho_0 \subseteq G \times G$, we want to know whether there is a bisimulation equivalence ρ such that $\rho_0 \subseteq \rho$. This is a special case of a classical problem well-studied in the literature; since our relations R_i are partial functions, we can give an alternative solution in a congruence closure style: our ‘Pseudo Congruence Closure’ (PCC) algorithm first computes the smallest equivalence relation ρ extending ρ_0 and then exhaustively applies to it the two rules of Fig. 1. The following Proposition is clear:

Proposition 4. *There is a bisimulation equivalence extending a relation ρ_0 on a finite Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ iff PCC, applied to ρ_0 , terminates without failure. In particular, for $g_1, g_2 \in G$ we have that $g_1 \sim_b g_2$ iff PCC does not fail on input $\langle \mathcal{G}, \rho_0 = \{(g_1, g_2)\} \rangle$.*

Remark 3. PCC can be simulated by an ordinary congruence closure problem (thus, PCC inherit the complexity bound $O(n \log n)$ [9, 17, 18] of congruence closure). To see this, it is sufficient to consider the signature comprising unary function symbols

F_i ($i \geq 0$) and unary predicates P_f (varying f among the function and constant symbols of Σ). Notice that such a signature is infinite but only finitely many symbols are used when handling a finite Σ -graph. Then consider the congruence closure problem given by the following literals: $g_1 = g_2$ (varying $(g_1, g_2) \in \rho_0$), $F_i(g) = g'$ (for $gR^i g'$ in \mathcal{G}), $P_f(g)$ (if $\lambda(g) = f$), $\neg P_f(g)$ (if $\lambda(g) \neq f$). Now it is easy to see that, despite the fact that the R^i are partial and the F_i are total functions, PCC and standard congruence closure run in the same way.

An important (infinite) Σ -graph is the Σ -graph of all rational trees. This is the Σ -graph $Rat = (\mathcal{R}, \{\mathcal{R}^i\}_i, \lambda_{\mathcal{R}})$, whose underlying set is formed by the set of rational trees \mathcal{R} and where we have

- $(T, \Lambda)R^i(T', \Lambda')$ iff (T', Λ') is the i -th successor of T (i.e. it is the i -th T -successor of the root subtree $T_\varepsilon = T$ of T);
- $\lambda_{\mathcal{R}}(T, \Lambda) = \Lambda(\varepsilon)$ (i.e. the label of (T, Λ) is the label of the root of T).

Inside Rat , bisimulation is trivial:

Proposition 5. *In the Σ -graph Rat , we have $(T, \Lambda) \sim_b (T', \Lambda')$ iff $(T, \Lambda) = (T', \Lambda')$*

Proof. Suppose that $(T, \Lambda) \sim_b (T', \Lambda')$. Recall from Remark 1 that if (T, Λ) is a Σ -tree, we write $(\sigma, f) \in (T, \Lambda)$ to mean that $\sigma \in T$ and $\Lambda(\sigma) = f$; moreover recall also that $(T, \Lambda) = (T', \Lambda')$ iff we have $(\sigma, f) \in (T, \Lambda)$ iff $(\sigma, f) \in (T', \Lambda')$ for all (σ, f) . This is what we are going to prove by induction on the length of σ .

If $\sigma = \varepsilon$, then from $(T, \Lambda) \sim_b (T', \Lambda')$ it follows that the root-labeling symbols of (T, Λ) and (T', Λ') are the same.

Suppose now that $\sigma = i * \tau$ and that $(i * \tau, f) \in (T, \Lambda)$; then $\Lambda(\varepsilon)$ is a function symbol of arity bigger or equal to i , so that there is the i -th successor (T_i, Λ_i) of (T, Λ) and we have $(T, \Lambda)\mathcal{R}^i(T_i, \Lambda_i)$ and $(\tau, f) \in (T_i, \Lambda_i)$. Since (T, Λ) and (T', Λ') are bisimilar, the symbols labeling their roots are the same, so for the i -th successor (T'_i, Λ'_i) of (T', Λ') we have that $(T', \Lambda')\mathcal{R}^i(T'_i, \Lambda'_i)$ and also that $(T_i, \Lambda_i) \sim_b (T'_i, \Lambda'_i)$ (again by bisimilarity). By induction hypothesis, $(\tau, f) \in (T'_i, \Lambda'_i)$, that is $(i * \tau, f) \in (T', \Lambda')$ as required. \dashv

The importance of Rat is due to the fact that every finite Σ -graph compares with it:

Proposition 6. *For every finite Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ there is a p -morphism $p_{\mathcal{G}} : \mathcal{G} \longrightarrow Rat$.*

Proof. Given $g \in G$, let us define a rational tree $p_{\mathcal{G}}(g)$ by specifying under which conditions we have $(\sigma, f) \in p_{\mathcal{G}}(g)$ for a finite list of positive numbers σ and $f \in \Sigma$ (recall Remark 1). We stipulate that $(i_1 \cdots i_k, f) \in p_{\mathcal{G}}(g)$ iff there are $g_1, \dots, g_k \in G$ such that $g\rho g_1 \cdots \rho g_k$ and $\lambda(g_k) = f$ (for $k = 0$, this means that $\lambda(g) = f$). From this definition, it is easy to see that $p_{\mathcal{G}}(g)$ is indeed a Σ -tree and that $p_{\mathcal{G}}$ is a p -morphism. The fact that $p_{\mathcal{G}}(g)$ is rational follows from the p -morphism condition, because the Σ -subtrees of $p_{\mathcal{G}}(g)$ turn out to be of the form $p_{\mathcal{G}}(g')$ for $g' \in G$ and G is finite. \dashv

We now relate Σ -graphs and constraint satisfiability in \mathcal{R} . We know from Proposition 2 that in order to check satisfiability of primitive formulae we can restrict to primitive formulae in solved form. To any constraint in solved form $\phi(x_1, \dots, x_n)$ we associate a Σ -graph

$$\mathcal{G}_\phi = (G_\phi, \{R_\phi^i\}_i, \lambda_\phi)$$

as follows. The nodes in G_ϕ are x_1, \dots, x_n ; we have $x_k R^i x_j$ iff ϕ contains an equality of the kind $x_k = f(\dots, x_j, \dots)$ (with x_j as i -th argument) and in such a case we put $\lambda_\phi(x_k) = f$. If ϕ contains an equality of the kind $x_k = a$ for a constant a , we put $\lambda_\phi(x_k) = a$. Notice that an equality like $x_k = f(\dots, x_j, \dots)$ or $x_k = a$ is uniquely identified for a variable x_k according to Definition 2. The variables x_k for which there are no equalities of the kind $x_k = f(\dots)$ or $x_k = a$ in ϕ are called *external* in ϕ . For such x_k we put $\lambda_\phi(x_k) = a_k$, where the a_k are *fresh distinct constants* in Σ (recall that Σ has infinitely many constants).

Theorem 1. *A constraint $\phi(x_1, \dots, x_n)$ in solved form is \mathcal{R} -satisfiable iff ϕ does not contain a disequality $x_k \neq x_j$ such that we have $x_k \sim_b x_j$ in \mathcal{G}_ϕ .*

Proof. Consider the p -morphism $p_{\mathcal{G}_\phi} : \mathcal{G} \rightarrow \text{Rat}$ of Proposition 6 and let us assign $p_{\mathcal{G}_\phi}(x_k)$ to every variable x_k occurring in ϕ ; notice that this \mathcal{R} -assignment (called the *canonical \mathcal{R} -assignment*) satisfies all equalities and all atomicity statements in ϕ (but it might not satisfy disequalities).

Suppose that there is no disequality $x_k \neq x_j$ in ϕ such that we have $x_k \sim_b x_j$ in \mathcal{G}_ϕ . If $x_k \neq x_j$ occurs in ϕ , we cannot have $p_{\mathcal{G}_\phi}(x_k) = p_{\mathcal{G}_\phi}(x_j)$, otherwise from $x_k \sim_b p_{\mathcal{G}_\phi}(x_k) = p_{\mathcal{G}_\phi}(x_j) \sim_b x_j$ we would get $x_k \sim_b x_j$ (this is because p -morphisms and their converses are bisimulations and bisimulations do compose). So the canonical \mathcal{R} -assignment is indeed a satisfying assignment for ϕ .

Vice versa, suppose that ϕ is satisfiable. By the next lemma, the canonical \mathcal{R} -assignment is a satisfying assignment for ϕ . If we have $x_k \sim_b x_j$ for an inequality $x_k \neq x_j$ occurring in ϕ , then we have that $p_{\mathcal{G}_\phi}(x_k) \sim_b p_{\mathcal{G}_\phi}(x_j)$ (because p -morphisms are bisimulations and bisimulations do compose) which contradicts $p_{\mathcal{G}_\phi}(x_k) \neq p_{\mathcal{G}_\phi}(x_j)$ and Proposition 5. \dashv

Lemma 1. *If a constraint $\phi(x_1, \dots, x_n)$ in solved form is \mathcal{R} -satisfiable, then it is satisfied by the canonical \mathcal{R} -assignment.*

Proof. Let α be a satisfying \mathcal{R} -assignment for ϕ and let α_C be the canonical \mathcal{R} -assignment. Since the latter can only fail to satisfy the disequalities from ϕ , it is sufficient to prove that for $x_k, x_j \in \{x_1, \dots, x_n\}$, we have $\alpha(x_i) \neq \alpha(x_j) \Rightarrow \alpha_C(x_i) \neq \alpha_C(x_j)$. In other words, we prove that *if there is (σ, f) such that $(\sigma, f) \in \alpha(x_k)$ and $(\sigma, f) \notin \alpha(x_j)$, then $\alpha_C(x_k) \neq \alpha_C(x_j)$* . We make an induction on the length of such σ .

Independently on the inductive argument, we notice that if either x_k or x_j (or both) is external in ϕ , the claim is trivial because external variables are mapped by α_C into singleton trees labelled by fresh distinct constants by the construction of \mathcal{G}_ϕ . If x_k, x_j are both non external, in ϕ there are equalities $x_k = l(v_1, \dots, v_m)$ and $x_j = l(w_1, \dots, w_m)$ with $v_1, \dots, v_m, w_1, \dots, w_m \in \{x_1, \dots, x_n\}$ (in the case where in ϕ there are equalities $x_k = l(\dots)$ and $x_j = l'(\dots)$ with $l \neq l'$, we again trivially have $\alpha_C(x_k) \neq \alpha_C(x_j)$ because α_C satisfies the equalities in ϕ).

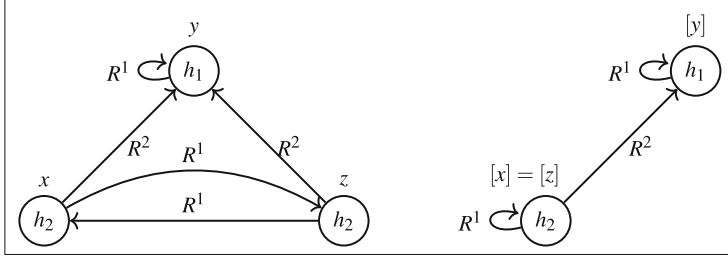


Fig. 2. Σ -graph from Example 1 (left) and its quotient under maximum bisimulation (right).

Let us now argue by induction on σ and restrict to the case where x_k, x_j are as above. The case $\sigma = \varepsilon$ is trivial because in that case we have $f = l$ and (ε, l) belongs to both $\alpha(x_k)$ and $\alpha(x_j)$, contrary to the fact that we supposed $(\sigma, f) \in \alpha(x_k)$ and $(\sigma, f) \notin \alpha(x_j)$ (α is a satisfying assignment, so $(\varepsilon, l) \in \alpha(x_j)$). So assume that $\sigma = i * \tau$; then $(i * \tau, f) \in \alpha(x_k)$ implies $(\tau, f) \in \alpha(v_i)$ because α satisfies $x_k = g(v_1, \dots, v_m)$. For the same reason, we have $(\tau, f) \notin \alpha(w_i)$, because otherwise $(i * \tau, f) \in \alpha(x_j)$ because α satisfies $x_j = g(w_1, \dots, w_m)$. We can then apply induction to (τ, f) and conclude that $\alpha_C(v_i) \neq \alpha_C(w_i)$. Since α_C nevertheless satisfies the equalities from ϕ , from the facts that $x_k = g(v_1, \dots, v_m)$ and $x_j = g(w_1, \dots, w_m)$ are true under α_C and that $\alpha_C(v_i) \neq \alpha_C(w_i)$, we conclude that $\alpha_C(x_k) \neq \alpha_C(x_j)$. \dashv

Remark 4. As a consequence of Theorem 1 and Proposition 4, in order to solve a \mathcal{R} -constraint satisfiability problem for a constraint $\phi(\underline{x})$ in solved form, it is sufficient to run PCC on inputs $\langle \mathcal{G}_\phi, \rho_0 = \{(x_1, x_2)\} \rangle$ for every disequality $x_1 \neq x_2$ occurring in ϕ . **The constraint is satisfiable iff PCC ends in failure for all such disequalities.** This gives a $O(n^2 \log n)$ complexity bound for constraints in solved form. As an alternative, one can directly compute the maximum bisimulation equivalence on the associated Σ -graph using some known efficient procedure [5–7].

Example 1. [In all examples, we assume that we have only two constructors: h_1 (with arity 1) and h_2 (with arity 2)]. Consider the constraint

$$x = h_2(z, y) \wedge y = h_1(y) \wedge z = h_2(x, y) \wedge x \neq y \wedge x \neq z$$

whose Σ -graph is depicted in Fig. 2. Nodes x and y are not bisimilar in this Σ -graph, but the nodes x and z are bisimilar. Hence the constraint is unsatisfiable. \dashv

5 Definability

In this section we investigate what it means for a set of variables to be ‘definable’ (via selectors and simultaneous parametric fixpoints) from the free variables \underline{x} of an existential formula $\exists y \phi(y, \underline{x})$. Definable and undefinable variables will be subject to different symbolic manipulations when converting an existential formula to a universal one (the possibility of such conversion is precisely the content of model completeness, see Definition 1).

Below, when we say that “ ϕ is $\phi' \wedge \psi$ ” we mean that ϕ is a conjunction and that it can be written as specified (modulo associativity and commutativity of \wedge).

Definition 6. Let $\exists \underline{y} \phi(\underline{y}, \underline{x})$ be an existential formula with free variables \underline{x} and bounded variables \underline{y} . The set of definable variables of $\exists \underline{y} \phi(\underline{y}, \underline{x})$ is the smallest subset $D \subseteq \underline{x} \cup \underline{y}$ satisfying the following conditions:

- (o) $\underline{x} \subseteq D$;
- (i) if $u \in D$ and ϕ is $\phi' \wedge u = t(\underline{v})$ for a proper flat term t , then $\underline{v} \subseteq D$;
- (ii) if $\underline{u} \subseteq D$ and ϕ is $\phi' \wedge \underline{v} = t(\underline{v}, \underline{u})$ for proper flat terms t , then $\underline{v} \subseteq D$.

Intuitively, the condition of Definition 6(i) says that the \underline{v} are reachable from u via selectors, whereas the condition of Definition 6(ii) says that the \underline{v} are reachable via some fixpoints having the \underline{u} as parameters. The next couple of lemmas show how to handle definable and non definable variables: the idea is that definable variables can be converted into universally quantified variables and that non definable variables can be removed because of validity/invalidity reasons.

Lemma 2. Suppose that the existential formula π is of the kind

$$\exists \underline{w} \exists \underline{u} \phi(\underline{w}, \underline{u}, \underline{x})$$

and that the variables \underline{u} are definable in it. The π is \mathcal{R} -equivalent to a formula of the kind

$$\forall \underline{u}' (\psi(\underline{u}', \underline{x}) \wedge (\theta(\underline{u}', \underline{x}) \rightarrow \exists \underline{w} \phi(\underline{w}, \underline{u}, \underline{x}))), \tag{4}$$

where $\underline{u}' \supseteq \underline{u}$ (i.e. \underline{u}' possibly extends \underline{u} by some fresh variables) and ψ, θ are quantifier-free. In particular, π is \mathcal{R} -equivalent to a universal formula in the case where $\underline{w} = \emptyset$ (i.e. in the case where all quantified variables of π are definable).

Proof. We view Definition 6 as a recursive definition and use the equivalences

$$[\exists \underline{v}' (u = t(\underline{v}, \underline{v}') \wedge \phi')] \leftrightarrow [\mathcal{R}_{h_i}(u) \wedge \forall \underline{w} \forall \underline{v}' (u = t(\underline{w}, \underline{v}') \rightarrow \underline{v} = \underline{w} \wedge \phi')] \tag{5}$$

(where t is a proper flat term whose root symbol is the constructor h_i) and

$$[\exists \underline{v} (\underline{v} = t(\underline{v}, \underline{u}) \wedge \phi')] \leftrightarrow [\forall \underline{v} (\underline{v} = t(\underline{v}, \underline{u}) \rightarrow \phi')] \tag{6}$$

Syntactic details are straightforward (they can be found in the online available extended version of the paper). ⊣

Example 2. Consider the formula

$$\exists w_1 \exists w_2 \exists w'_1 \exists w'_2 \exists u (x_1 = h_2(w_1, w_2) \wedge x_2 = h_2(w'_1, w'_2) \wedge u = h_1(w_1) \wedge u \neq w_1) \tag{7}$$

whose existentially quantified variables are all definable. In particular, the variables w_1, w_2, w'_1, w'_2 falls within case (i) of Definition 6, hence their conversion into universal variables follows the schema (5). On the contrary, u is converted into a universal variable using schema (6), because u falls within the case (ii) of Definition 6: in fact, u is recursively definable from w_1 via the equality $u = h_1(w_1)$ (the latter is a fixpoint equations - indeed a trivial fixpoints equation where the variables on the left hand side of the equality symbol do not occur on the right hand side term). ⊣

Lemma 3. *Suppose that $C(\underline{x}, \underline{y})$ is a constraint in solved form and that in each literal from C there is at least one occurrence of a variable from \underline{y} ; suppose also that in the existential formula $\exists \underline{y} C(\underline{x}, \underline{y})$ the variables \underline{y} are not definable. Then we have*

$$\mathcal{R} \models \forall \underline{x} (\text{Distinct}(\underline{x}) \rightarrow [\exists \underline{y} C(\underline{x}, \underline{y}) \leftrightarrow \exists \underline{x} \exists \underline{y} C(\underline{x}, \underline{y})]) \quad (8)$$

(here $\text{Distinct}(\underline{x})$ means $\bigwedge (x' \neq x'')$, varying the conjuncts on pairwise different $x', x'' \in \underline{x}$).

Proof. First a comment: what the lemma says is that, in contexts where all parameters \underline{x} are interpreted as distinct trees, $\exists \underline{y} C(\underline{x}, \underline{y})$ is equivalent to the sentence $\exists \underline{x} \exists \underline{y} C(\underline{x}, \underline{y})$, which in turn is always equivalent to either \top or \perp by Theorem 1.

The implication from left to right of \leftrightarrow is trivial, so let us assume that in the rational tree structure \mathcal{R} the sentence

$$\exists \underline{x} \exists \underline{y} C(\underline{x}, \underline{y}) \quad (9)$$

is true and let us assign to $\underline{x} = x_1, \dots, x_n$ arbitrary but distinct rational trees $\alpha(x_1), \dots, \alpha(x_n)$. We want to show that this assignment satisfies the formula $\exists \underline{y} C(\underline{x}, \underline{y})$. Since our trees are rational, the trees $\alpha(x_1), \dots, \alpha(x_n)$ have only finitely many subtrees; let their number be $n + n'$ and let us list them as

$$\alpha(x_1), \dots, \alpha(x_n), \alpha(x'_1), \dots, \alpha(x'_{n'}) . \quad (10)$$

We now write down a constraint $C'(\underline{x}, \underline{x}')$ (where $\underline{x}' = x'_1, \dots, x'_{n'}$) whose *unique* solution is precisely the $n + n'$ -tuple of trees (10).² We shall show that the constraint

$$\exists \underline{x} \exists \underline{x}' \exists \underline{y} (C(\underline{x}, \underline{y}) \wedge C'(\underline{x}, \underline{x}')) \quad (11)$$

is satisfiable; this proves our claim, because the satisfiability of (11) implies in particular that $\alpha(x_1), \dots, \alpha(x_n)$ satisfy $\exists \underline{y} C(\underline{x}, \underline{y})$. We now build the Σ -graphs

$$\mathcal{G}_{(9)} = (G_{(9)}, \{R_{(9)}^i\}_i, \lambda_{(9)}) \quad \text{and} \quad \mathcal{G}_{(11)} = (G_{(11)}, \{R_{(11)}^i\}_i, \lambda_{(11)})$$

associated with the formulae (9) and (11), respectively. In both cases, *when choosing the constants labeling external variables, we take fresh constants* not appearing in (9) and (11): we can do that because our signature Σ contains infinitely many constants and the trees $\alpha(x_1), \dots, \alpha(x_n)$ are rational.

We make a couple of observations. Since the \underline{y} occur in each literal from C and the \underline{y} are not definable, the equality flat literals from C must be of the kind $v = t(\underline{x}, \underline{y})$ where $v \in \underline{y}$, so *the \underline{x} are not head variables in C* . Moreover, *every head variable $v \in \underline{y}$ has a path³ to an external variable $w \in \underline{y}$ in $\mathcal{G}_{(9)}$* (and consequently also in the bigger Σ -graph $\mathcal{G}_{(11)}$): if it were not be so, considering the equality proper flat literals from C reachable

² For instance, if $\alpha(x_1)$ is rooted by h_2 and has immediate successors the pair formed by $\alpha(x'_4), \alpha(x_1)$, then C' contains as a conjunct the equality $x_1 = h_2(x'_4, x_1)$, etc. Here we are using Proposition 1(vi) (with an empty parameters tuple \underline{z}).

³ By a path in a Σ -graph $\mathcal{G} = (G, \{R^i\}_{i \geq 1}, \lambda)$ from a node $g \in G$ to a node $g' \in G$ we mean a chain of the kind $g = g_0 R^{i_1} g_1 \cdots g_{n-1} R^{i_n} g_n = g'$.

from v , we could obtain a set of proper flat equalities witnessing the definability of a subset of variables that includes v (see Definition (6)(ii)). By the equality proper flat literals *reachable from* v , we mean the smallest set of literals from C containing the equality proper flat literal headed by v and such that if it contains $v' = t(\underline{u}, \underline{x})$, then for every $u \in \underline{u}$ it contains also the equality proper flat literal headed by u (if it exists).

Let us now consider a bisimulation relation in $\mathcal{G}_{(11)}$: we claim that this must be of the kind $\rho \cup id_{G_{(11)}}$, where ρ is a set of pairs (v_1, v_2) formed by nodes which are both head nodes belonging to \underline{y} . In fact, external nodes from \underline{y} are not bisimilar to each other (they are labeled by different constants) nor can be bisimilar to the $\underline{x}, \underline{x}'$ (the constants labeling them are disjoint from the constants labeling the $\underline{x}, \underline{x}'$) nor to head nodes from \underline{y} (the latter are not labeled by constants). Similarly, the nodes $\underline{x}, \underline{x}'$ are not bisimilar to each other (the sub- Σ -graph of $\mathcal{G}_{(11)}$ involving the $\underline{x}, \underline{x}'$ is a sub- Σ -graph of *Rat* - where the only bisimulation is identity, see Proposition 5 - recall that the $\underline{x}, \underline{x}'$ represent in $\mathcal{G}_{(11)}$ the pairwise different trees (10)) nor to head nodes from \underline{y} (because the latter have a path to some external node taken from the \underline{y} , as explained above, and the $\underline{x}, \underline{x}'$ only have paths inside themselves).

Suppose now that (11) is not satisfiable; this means that there is a disequality $u_1 \neq u_2$ from C (recall that C' does not contain disequalities) such that PCC does not fail when initialized to $\rho = \{(u_1, u_2)\}$ in $G_{(11)}$. Since this entails that u_1 and u_2 are bisimilar in $G_{(11)}$, we have that u_1, u_2 are both head nodes belonging to \underline{y} . By induction, we show that PCC initialized to $\rho = \{(u_1, u_2)\}$ in $G_{(9)}$ makes precisely the same steps and halts precisely after the same number of steps as PCC initialized to $\rho = \{(u_1, u_2)\}$ in $G_{(11)}$ (this would mean that (9) is not satisfiable either, a contradiction). Suppose in fact that PCC in $G_{(11)}$ added to ρ a pair of head nodes v_1, v_2 belonging to \underline{y} (only such nodes can be bisimilar and not identical to each other) and that we have $v_1 R_{G_{(11)}}^i v'_1, v_2 R_{G_{(11)}}^i v'_2$. Then v'_1 and v'_2 are bisimilar and so, if they are not identical to each other, they are also head nodes belonging to \underline{y} . In such a case, we have $v_1 R_{G_{(9)}}^i v'_1, v_2 R_{G_{(9)}}^i v'_2$ and so PCC in $G_{(9)}$ merges their equivalence classes precisely as PCC does in $G_{(11)}$. Moreover, when PCC halts in $G_{(11)}$, so must do PCC in $G_{(9)}$ because nodes in \underline{y} can see via the R^i the same nodes in both graphs: thus if PCC halted in $G_{(11)}$ and in $G_{(9)}$ for $(v_1, v_2) \in \rho$ we have $v_1 R_{G_{(9)}}^i v'_1, v_2 R_{G_{(9)}}^i v'_2$, then we have also $v_1 R_{G_{(11)}}^i v'_1, v_2 R_{G_{(11)}}^i v'_2$ in $G_{(11)}$ and (v'_1, v'_2) has been already added by PCC to ρ (actually in both Σ -graphs) because PCC halted in $G_{(11)}$. ⊥

6 Model Completeness

We improve Proposition 2. Let us say that a primitive formula is in *reduced solved form* iff it can be written in the form $E(\underline{x}', \underline{x}) \wedge \exists \underline{y} C(\underline{x}, \underline{y})$ (see (3)) and, in addition to the requirements of Definition 3, we also ask that the variables \underline{y} are definable in $\exists \underline{y} C(\underline{x}, \underline{y})$.

Theorem 2. *Every existential formula is \mathcal{R} -equivalent to a disjunction of primitive formulae in reduced solved form and also to a universal formula. As a consequence, \mathcal{R} is model-complete.*

Proof. Consider an existential formula $\pi(\underline{x})$ of the form $\exists \underline{w} \phi(\underline{y}, \underline{w})$. Given an equivalence relation E over $\underline{v} \cup \underline{w}$, let us call ‘full display of E ’ the conjunction of all the

equalities $z_1 = z_2$ (for $(z_1, z_2) \in E$) and of all the disequalities $z_1 \neq z_2$ (for $(z_1, z_2) \notin E$). We conjoin the matrix $\phi(\underline{v}, \underline{w})$ of our existential formula $\exists \underline{w} \phi(\underline{v}, \underline{w})$ with the disjunction over the full displays of all the possible equivalence relations over $\underline{v} \cup \underline{w}$; then we take DNF, distribute $\exists \underline{w}$ over disjunctions and leave each disjunct be handled by the algorithm of Proposition 2. As a result, our π is equivalent to a disjunction of primitive solved forms $E(\underline{x}', \underline{x}) \wedge \exists \underline{y} C(\underline{x}, \underline{y})$, whose matrix $C(\underline{x}, \underline{y})$ is of the kind $\text{Distinct}(\underline{x}, \underline{y}) \wedge \phi(\underline{x}, \underline{y})$. We shall treat these disjuncts separately.

Fix then such a disjunct. Let us split \underline{y} as $\underline{y}', \underline{y}''$, where the \underline{y}' are definable and the \underline{y}'' are not definable in $\exists \underline{y} C(\underline{x}, \underline{y})$. Thus we can rewrite $\exists \underline{y} C(\underline{x}, \underline{y})$ as

$$\exists \underline{y}' (C'(\underline{x}, \underline{y}') \wedge \exists \underline{y}'' C''(\underline{x}, \underline{y}', \underline{y}'')) \quad (12)$$

where in C'' all literals contain at least one occurrence of some of the \underline{y}'' , the \underline{y}' are definable in $\exists \underline{y}' C'(\underline{x}, \underline{y}')$, the \underline{y}'' are not definable in $\exists \underline{y}'' C''(\underline{x}, \underline{y}', \underline{y}'')$ and the constraint C' contains the literals $\text{Distinct}(\underline{x}, \underline{y}')$ (this is because $\text{Distinct}(\underline{x}, \underline{y}', \underline{y}'')$ entails $\text{Distinct}(\underline{x}, \underline{y}')$). Now we can apply Lemma 3, according to which the subformula $\exists \underline{y}'' C''(\underline{x}, \underline{y}', \underline{y}'')$ can be replaced by its existential closure $\exists \underline{x} \exists \underline{y}' \exists \underline{y}'' C''(\underline{x}, \underline{y}', \underline{y}'')$ and the latter simplifies either to \top or to \perp .

In conclusion, our initial existential formula $\pi(\underline{x})$ is equivalent to a formula of the required shape, namely to a disjunction of primitive formulae in solved form

$$\bigvee_k \left[E_k(\underline{x}'_k, \underline{x}_k) \wedge \exists \underline{y}_k C_k(\underline{y}_k, \underline{x}_k) \right], \quad (13)$$

where the \underline{y}_k are definable in $\exists \underline{y}_k C_k(\underline{y}_k, \underline{x}_k)$ (these are the disjuncts surviving after the above simplifications). Such a formula is also equivalent to a universal formula by Lemma 2. \dashv

Theorem 2 (together with Theorem 1) *gives an algorithm for deciding \mathcal{R} -satisfiability of any first-order sentence*, because this theorem supplies an effective procedure to rewrite an existential formula to a universal one (see the remark after Definition 1).

Example 3. Consider the formula

$$\exists w_1 \exists w_2 \forall z \forall u (x_1 = h_2(w_1, w_2) \wedge x_2 \neq h_1(z) \wedge \neg \text{At}(x_2) \wedge \neg (h_1(u) = u \wedge u = w_2)) . \quad (14)$$

To bring it to the form (13) (and then check its satisfiability), we first rewrite it as $\exists w_1 \exists w_2 \neg \exists z \exists u \neg \psi$ (here ψ is the matrix of (14)), then convert $\exists z \exists u \neg \psi$ to a universal formula θ ; in this way $\exists w_1 \exists w_2 \neg \theta$ will be again existential. To the latter, we can apply the procedure of Theorem 2 and obtain a disjunction of primitive formulae in reduced solved form. After simplifications, in our case we get just one disjunct, namely

$$\exists w_1 \exists w_2 \exists w'_1 \exists w'_2 \exists u (x_1 = h_2(w_1, w_2) \wedge x_2 = h_2(w'_1, w'_2) \wedge u = h_1(w_1) \wedge u \neq w_1) \quad (15)$$

(this is the same formula (7) of Example 2 which is, by the way, satisfiable according to Theorem 1). \dashv

The proof of Theorem 2 shows how to build a definitional extension of \mathcal{R} enjoying quantifier elimination. To this aim, it is sufficient to add to the language new operation symbols for selectors and simultaneous parametric fixpoints.

For selectors, we need to take care of the fact that selectors are not totally defined. We adopt the classical solution of [11], saying that a badly applied selector just returns its argument. In other words, for every constructor h_i of arity ar_i , we add to the signature of \mathcal{R} new unary functions symbols Sel_i^j (for $j = 1, \dots, ar_i$) to be interpreted in rational trees so that the following formula is true for every assignment to x, y :

$$\text{Sel}_i^j(x) = y \leftrightarrow (\exists z_1 \dots \exists z_{ar_i} (x = h_i(z_1, \dots, z_{ar_i}) \wedge y = z_j)) \vee (\neg R_{h_i}(x) \wedge y = x) \quad (16)$$

For simultaneous parametric fixpoints, we need infinitely many extra functions. Consider two tuples of variables $\underline{x}^- = x_1^-, \dots, x_n^-$ and $\underline{y}^* = y_1^*, \dots, y_m^*$ and proper flat literals $\underline{x}^- = \underline{t}(\underline{x}^-, \underline{y}^*)$: below, we write $\underline{t}(-, *)$ for $\underline{t}(\underline{x}^-, \underline{y}^*)$ and $\underline{t}(\underline{x}, \underline{y})$ for a substitution $\underline{t}(\underline{x}/\underline{x}^-, \underline{y}/\underline{y}^*)$. We introduce n new functions symbols $\text{Fix}_{\underline{t}(-, *)}^i$ ($i = 1, \dots, n$), all having arity m , interpreted in rational trees so that the following formula is true, for every assignment to the variable x and to the m -tuple of variables \underline{y} :

$$\text{Fix}_{\underline{t}(-, *)}^i(\underline{y}) = x \leftrightarrow \exists \underline{x} (\underline{x} = \underline{t}(\underline{x}, \underline{y}) \wedge x = x_i) \quad (17)$$

where x_i denotes the i -th component of the tuple \underline{x} .

Let us denote with \mathcal{R}^+ the Σ -structure of rational trees, enriched with the interpretation of the above extra function symbols; we also denote by \mathcal{R}^+ the theory whose axioms are the sentences which are true in this expanded structure. From Theorem 2 and (16), (17), we have the following result (see the online available extended version of the paper for the straightforward syntactic details):

Theorem 3. \mathcal{R}^+ enjoys quantifier elimination.

Example 4. The formula $R_{h_i}(x)$ (that can be written both as a universal and as an existential formula in \mathcal{R}) can be rewritten in \mathcal{R}^+ as

$$x = h_i(\text{Sel}_i^1(x), \dots, \text{Sel}_i^{ar_i}(x)) \quad (18)$$

without using quantifiers. ⊣

Example 5. Eliminating quantifiers from (15), we obtain

$$x_1 = h_2(\text{Sel}_{h_2}^1(x_1), \text{Sel}_{h_2}^2(x_1)) \wedge x_2 = h_2(\text{Sel}_{h_2}^1(x_2), \text{Sel}_{h_2}^2(x_2)) \wedge \text{Sel}_{h_2}^1(x_1) \neq \text{Fix}_{h_1(-)}^1; \quad (19)$$

this formula says that x_1, x_2 are both rooted with h_2 and that applying the first selector to x_1 we get something different from the fixed point of h_1 ;⁴ the latter can be represented as the infinite term

$$h_1(h_1(h_1 \dots$$

(it is the infinite unary tree labelled by h_1). ⊣

⁴ Formally, $\text{Fix}_{h_1(-)}^1$ is the constant expressing the fixpoint of $z = h_1(z)$: this is the simultaneous fixpoint of a tuple of functions with empty parameters formed by the single function $h_1(z)$ (the parameters $*$ are missing and the superscript 1 means the projection to the first unique component of such tuple of functions).

7 Conclusions, Related and Further Work

In this paper we introduced the variant of the theory of finite and infinite trees, whose signature has finitely many constructors, infinitely many constants and an atomicity predicate. We proved that this theory is model complete, showed that every formula in this theory is equivalent to a disjunction of primitive formulae in reduced solved form and proved that one can achieve full quantifier elimination by adding selectors and simultaneous parametric fixpoints to the language.

One of the main insights given by the above results is the fact that every formula of our theory essentially expresses a Boolean combination of *algebraic dependency relations involving constructors, selectors and fixpoints* (see Example 5 to understand what we mean). We believe that this is the contribution lying behind statements like those of Theorems 2, 3. The ‘explicit solved forms’ of [4] and the ‘terms with pointers’ of [14] ultimately carry on this information too, but maybe in a less transparent way (a thorough comparison requires technical details, we cannot report them here for space reasons, but see the online available extended version of the paper).

One may wonder whether our analysis extends to other variants of the theory of trees. Clearly, if for instance we have infinitely many constructors in the signature, we lose the possibility of expressing formulae like $R_{h_i}(x)$ both as existential and universal formulae (see (1) and (2)), so model-completeness is likely to fail. Other partial results might probably be recovered, this has to be investigated by future work.

We wonder whether our techniques can be effective also for extensions of the theory of trees, we leave this too for future work. For instance, adding a finiteness predicate (like in [4, 22]) is worth investigating.

Concerning our algorithms, we underline that the constraint solving algorithm of Sect. 4 (based on bisimulations and congruence closure) is rather efficient and its complexity is comparable with analogous constraint solving algorithms from the literature [19]. The situation is different for our decision procedure once extended to all elementary formulae. We believe that the algorithm of Theorem 2, relying on model-completeness, is direct and intuitive, however important improvements still need to be designed.

A first improvement should avoid any guessing of an equivalence relation between variables in the proof of Theorem 2: this improvement however requires some care and involves a strengthening of the technical Lemma 3, so we preferred to leave it for future work. The strengthening should be able to compute a disjunction of e-diagrams over \underline{x} equivalent to $\exists \underline{y} C(\underline{x}, \underline{y})$, for every existential formula $\exists \underline{y} C(\underline{x}, \underline{y})$ satisfying the peculiar hypotheses of Lemma 3.

The other source of complexity of the algorithm deciding satisfiability of all first-order formulae is the need of DNF conversions every time a quantifier alternation is removed. Although this problem is somewhat unavoidable (see the lower bound mentioned in the introduction), we believe that many redundancies arising during computations can be removed with suitable heuristics and simplification routines. Another possibility is that of exploiting the rich algebraic structure of \mathcal{R}^+ in order to directly design a quantifier elimination algorithm in \mathcal{R}^+ : here the rich algebraic structure should consent the adoption of ‘testing points’ methods, similar to those employed in quantifier elimination for numerical domains [1].

References

1. Bockmayr, A., Weispfenning, V.: Solving numerical constraints. In: *Handbook of Automated Reasoning*, vol. II, pp. 751–844. Elsevier and MIT Press (2001)
2. Chang, C.-C., Keisler, J.H.: *Model Theory*, 3rd edn. North-Holland Publishing Co., Amsterdam-London (1990)
3. Colmerauer, A.: Equations and inequations on finite and infinite trees. In: *Fifth Generation Computer Systems* (1984)
4. Djelloul, K., Dao, T., Frühwirth, T.W.: Theory of finite or infinite trees revisited. *Theory Pract. Logic Program.* **8**(4), 431–489 (2008)
5. Dovier, A., Piazza, C., Policriti, A.: A fast bisimulation algorithm. In: Berry, G., Comon, H., Finkel, A. (eds.) *CAV 2001*. LNCS, vol. 2102, pp. 79–90. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_8
6. Gentilini, R., Piazza, C., Policriti, A.: Simulation as coarsest partition problem. In: Katoen, J.-P., Stevens, P. (eds.) *TACAS 2002*. LNCS, vol. 2280, pp. 415–430. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_29
7. Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation: coarsest partition problems. *J. Autom. Reason.* **31**, 73–103 (2003)
8. Jaffar, J.: Efficient unification over infinite terms. *N. Gener. Comput.* **2**, 207–219 (1984)
9. Kapur, D.: Shostak’s congruence closure as completion. In: Comon, H. (ed.) *RTA 1997*. LNCS, vol. 1232, pp. 23–37. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62950-5_59
10. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: Young, M., Devanbu, P.T. (eds.) *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, 5–11 November 2006*, pp. 105–116. ACM (2006)
11. Kunen, K.: Negation in logic programming. *J. Logic Program.* **4**, 289–308 (1987)
12. Maher, M.J.: Complete axiomatizations of the algebras of finite, rational and infinite trees. In: *Proceedings Third Annual Symposium on Logic in Computer Science*, pp. 348–349. IEEE Computer Society (1988)
13. Mal’cev, A.: On the elementary theory of locally free algebras. *Soviet Math. Doklady* 768–871 (1961)
14. Marongiu, G., Tulipani, S.: Quantifier elimination for infinite terms. *Arch. Math. Log.* **31**(1), 1–17 (1991)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: *Proceedings of CAV*, pp. 123–136 (2006)
16. Meister, M., Frühwirth, T.: Complexity of the CHR rational tree equation solver. In: *Proceedings of the Third Workshop on Constraint Handling Rules* (2006)
17. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. *J. ACM* **27**(2), 356–364 (1980)
18. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. *Inf. Comput.* **205**(4), 557–580 (2007)
19. Podelski, A., Roy, P.V.: The beauty and the beast algorithm: Quasi-linear incremental tests of entailment and disentailment over trees. In: *International Logic Programming Symposium (ILPS)*, pp. 359–374. MIT Press (1994)
20. Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. In: Kambhampati, S. (ed.) *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016*, pp. 4205–4209. IJCAI/AAAI Press (2016)

21. Vorobyov, S.: An improved lower bound for the elementary theories of trees. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 275–287. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61511-3_91
22. Zaiser, F., Ong, L.: Abstract: the extended theory of trees and algebraic (co)datatypes. In: Nadel, A., Niemetz, A. (eds.) Proceedings of the 19th International Workshop on Satisfiability Modulo Theories co-located with 33rd International Conference on Computer Aided Verification (CAV 2021), Online (initially located in Los Angeles, USA), 18–19 July 2021. CEUR Workshop Proceedings, vol. 2908, p. 65. CEUR-WS.org (2021)

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

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





Certifying Phase Abstraction

Nils Froleyks¹(✉), Emily Yu², Armin Biere³, and Keijo Heljanko^{4,5}

¹ Johannes Kepler University, Linz, Austria
N.froleyks@gmail.com

² Institute of Science and Technology Austria, Klosterneuburg, Austria

³ Albert–Ludwigs–University, Freiburg, Germany

⁴ University of Helsinki, Helsinki, Finland

⁵ Helsinki Institute for Information Technology, Helsinki, Finland

Abstract. Certification helps to increase trust in formal verification of safety-critical systems which require assurance on their correctness. In hardware model checking, a widely used formal verification technique, phase abstraction is considered one of the most commonly used preprocessing techniques. We present an approach to certify an extended form of phase abstraction using a generic certificate format. As in earlier works our approach involves constructing a witness circuit with an inductive invariant property that certifies the correctness of the entire model checking process, which is then validated by an independent certificate checker. We have implemented and evaluated the proposed approach including certification for various preprocessing configurations on hardware model checking competition benchmarks. As an improvement on previous work in this area, the proposed method is able to efficiently complete certification with an overhead of a fraction of model checking time.

1 Introduction

Over the past few decades, symbolic model checking [2, 22, 23] has been put forward as one of the most effective techniques in formal verification. A lot of trust is placed into model checking tools when assessing the correctness of safety-critical systems. However, model checkers themselves and the symbolic reasoning tools they rely on, are exceedingly complex, both in the theory of their algorithms and their practical implementation. They often run for multiple days, distributed across hundreds of interacting threads, ultimately yielding a single bit of information signaling the verification result. To increase trust in these tools, several approaches have attempted to implement fully verified model checkers in a theorem proving environment such as Isabelle [1, 27, 54]. However, the scalability as well as versatility of those tools is often rather limited. For example, a technique update tends to require the entire tool to be re-verified.

An alternative is to make model checkers provide machine-checkable proofs as certificates that can be validated by independent checkers [8–10, 31, 32, 39, 42, 47], which is already a successful approach in SAT [34, 35], i.e., proofs are mandatory in the SAT competition since 2016 [3], and they are a very hot topic

in SMT [4, 5, 36, 51] and beyond [4]. Crucially, these certificates need to be simple enough to allow the implementation of a fully verified proof checker [33, 37, 41], and preferably verifiable “end-to-end”, i.e., certifying all stages of the model checking process, including all forms of preprocessing steps.

The approach in [15, 56, 57] introduces a generic certificate format that can be directly generated from hardware model checkers via book-keeping. More specifically, the certificate is in the form of a Boolean circuit that comes with an inductive invariant, such that it can be verified by six simple SAT checks. So far, it has shown to be effective across several model checking techniques, but has not covered phase abstraction [16]. The experimental results from [15, 56, 57] also show performance challenges with more complex model checking problems. In this paper, we focus on refining the format for smaller certificates while accommodating additional techniques such as cone-of-influence analysis reduction [22].

Phase abstraction [16] is a popular preprocessing technique which tries to simplify a given model checking problem by detecting and removing periodic signals that exhibit clock-like behaviors. These signals are essentially the clocks embedded in circuit designs, often due to the design style of multi-phase clocking [46]. Phase abstraction helps reduce circuit complexity therefore making the backend model checking task easier. Differently from [6, 7] where the concept was first suggested, requiring syntactic analysis and user inputs, phase abstraction [16] makes use of ternary simulation to automatically identify a group of clock-like latches. Beside this, ternary simulation has also been utilized in the context of temporal decomposition [20] for detecting transient signals.

In industrial settings, due to the use of complex reset logic as well as circuit synthesis optimizations, clock signals are sometimes delayed by a number of initialization steps [19]. To further optimize the verification procedure we extend phase abstraction by exploiting the power of ternary simulation to capture different classes of periodic signals including those that are considered partially as clocks as well as equivalent signals [26]. An optimal phase number is computed based on globally extracted patterns, which then is used to unfold the circuit multiple times. The resulting unfolded circuit further undergoes rewriting and cone-of-influence reduction, before it is passed on to a base model checker for final verification. To summarize our contributions are as follows:

1. We formalize, revisit and extend the original phase abstraction [16] by introducing periodic signals, that are then identified and removed for circuit reduction. Our technique also subsumes temporal decomposition [20].
2. Building upon [15, 56, 57], we propose a refined certificate format for hardware model checking based on a new *restricted simulation* relation. We demonstrate how to build such a certificate for extended phase abstraction.
3. We present MC2, a certifying model checker that implements our proposed preprocessing technique and generates certificates for the entire model checking process. We show empirically that the approach requires small certification overhead in contrast to [15, 56, 57].

After background in Sect. 2, Sect. 3 introduces the notion of periodic signals. In Sect. 4 we present an extended variant of phase abstraction that simplifies

the original model with periodic signals. In Sect. 5 we define a refined certificate format and present a general certification approach for phase abstraction. In Sect. 6 we describe the implementation of MC2 and then show the effectiveness of our new certification approach in Sect. 7.

2 Background

Given a set of Boolean variables \mathcal{V} , a literal l is either a variable $v \in \mathcal{V}$ or its negation $\neg v$. A *cube* is considered to be a non-contradictory set of literals. Let c be such a cube over a set of variables L and assume L' are copies of L , i.e., each $l \in L$ corresponds bijectively to an $l' \in L'$. Then we write $c(L')$ to denote the resulting cube after replacing the variables in c with its corresponding variables in L' . For a Boolean formula f , we write $f|_l$ and $f|_{\neg l}$ to denote the formula after substituting all occurrences of the literal l with \top and \perp respectively. We use equality symbols \simeq [24] and \equiv to denote syntactic and semantic equivalence and similarly \rightarrow and \Rightarrow to denote syntactic and semantic logical implication.

Definition 1 (Circuit). *A circuit C is represented by a quintuple (I, L, R, F, P) , where I and L are (finite) sets of input and latch variables. The reset functions are given as $R = \{r_l(I, L) \mid l \in L\}$ where the individual reset function $r_l(I, L)$ for a latch $l \in L$ is a Boolean formula over inputs I and latches L . Similarly the set of transition functions is given as $F = \{f_l(I, L) \mid l \in L\}$. Finally $P(I, L)$ denotes a safety property corresponding to set of good states again encoded as a Boolean formula over the inputs and latches.*

This notion can be extended to more general circuits involving for instance word-level semantics or even continuous variables by replacing in this definition Boolean formulas by corresponding predicates and terms in first-order logic modulo theories. For simplicity of exposition we focus in this work on Boolean semantics, which matches the main application area we are targeting, i.e., industrial-scale gate-level hardware model checking. We claim that extensions to “circuits modulo theories” are quite straightforward.

A concrete state is an assignment to variables $I \cup L$. Therefore the set of reset states of a circuit is the set of satisfying assignments to $R(L) = \bigwedge_{l \in L} (l \simeq r_l(I, L))$.

Note the use of syntactic equality “ \simeq ” in this definition.

As in previous work [15] we assume acyclic reset functions. Therefore $R(L)$ is always satisfiable. A circuit with acyclic reset functions is called *stratified*.

As in bounded model checking [11], with I_i and L_i “temporal” copies of I and L at time step i , the *unrolling* of a circuit up to length k is expressed as:

$$U_k = \bigwedge_{i \in [0, k)} (L_{i+1} \simeq F(I_i, L_i)).$$

Cube simulation [57] subsumes ternary simulation such that a lasso found by ternary simulation can also be found via cube simulation. A cube simulation is a sequence of cubes $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$ over latches L such that (1) $R(L) \Rightarrow c_0$;

(2) $c_i \wedge (L' \simeq F(I, L)) \Rightarrow c'_{i+1}$ for all $i \in [0, \delta + \omega)$, where c'_{i+1} is the primed copy of c_{i+1} . It is called a cube lasso if $c_{\delta+\omega} \wedge (L' \simeq F(I, L)) \Rightarrow c'_\delta$. In which case δ is the stem length and ω is the loop length. For $\delta = 0$, the initial cube is already part of the loop and for $\omega = 0$, the lasso ends in a self-loop.

3 Periodic Signals

In sequential hardware designs, signals that eventually stabilize to a constant, i.e., to \top or \perp , after certain initialization steps are called *transient* signals [20, 57], whereas oscillating signals have clock-like or periodic behaviors. A simplest example of a clock is a latch that always oscillates between \top and \perp .

Since hardware designs typically consist of complex initialization logic, there are occurrences of delayed oscillating signals, like clocks that start ticking after several reset steps, with a combination of transient and clock behaviours. We generalize this concept to categorize latches as periodic signals associated with a *duration* (i.e., the number of time steps for which a signal is delayed) and a *phase number* (i.e., the period length in a periodic behavior). Moreover, our generalization also captures equivalent and antivalent signals [26], as well as those that exhibit partial periodic behaviours. See Fig. 1 for an example.

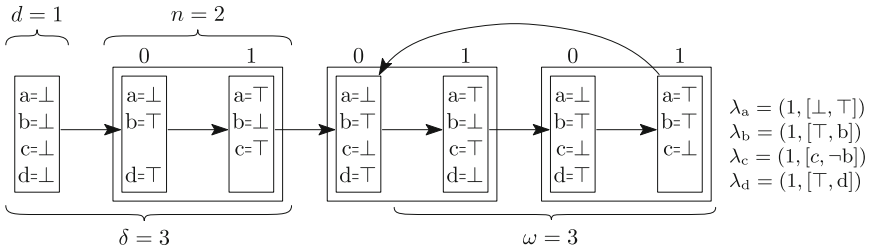


Fig. 1. An example of a cube lasso over the latches $l \in L = \{a, b, c, d\}$. In the example the tall rectangles represent cubes as partial assignments, i.e., the second cube from the left is $(\neg a) \wedge b \wedge d$. Phase 0 and 1 are marked on top of the cubes. As shown, duration $d = 1$ and phase number $n = 2$ yield a high number of useful signals for this cube lasso. Each latch l is associated with a periodic pattern λ_l on the right describing its behaviors for phase 0 and 1. If a latch is missing from a cube for a certain phase, it has no constraint thus we use the equality of the latch to itself in the signal. Latch a turns out to be a simple clock delayed by one step. Latches b and d behave clock-like but only in phase 0. Latch c always has the opposite value of latch b in phase 1. Note that we could also have $\neg c$ in phase 1 of signal λ_b but choosing a single representative for a set of equivalent signals is beneficial for the following simplification steps.

Definition 2 (Periodic Signal). Given a circuit $C = (I, L, R, F, P)$ and a cube lasso $c_0, \dots, c_\delta, \dots, c_{\delta+\omega}$. A periodic signal λ_l for a latch $l \in L$ is defined as $\lambda_l = (d, [v^0, \dots, v^{n-1}])$ where $d \in \mathbb{N}$, $n \in \mathbb{N}^+$ and v^i is a latch literal or a constant, with $d \leq \delta$. We further require that there exist $k^0, k^1 \in \mathbb{N}^+$ with

$k^0 \cdot n + d = \delta$ and $k^1 \cdot n = \omega + 1$ such that for all $i \in [0, n)$ and $j \in [0, k^0 + k^1)$ we have $c_{i+j \cdot n} \Rightarrow (l \simeq v^i)$.

For a signal $\lambda_l = (d, [v^0, \dots, v^{n-1}])$ we will write λ_l^i to refer to the i -th element of $[v^0, \dots, v^{n-1}]$, which we refer to as its phase. See Fig. 1 for an example where $k^0 = 1$ and $k^1 = 2$.

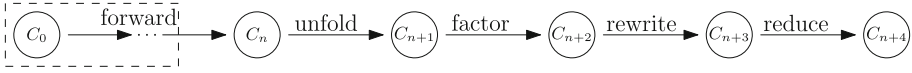


Fig. 2. Circuit transformation using phase abstraction.

4 Extending Phase Abstraction

In this section, we revisit and extend phase abstraction by defining it as a sequence of preprocessing steps, as illustrated in Fig. 2. Differently from the approach in [16], we present phase abstraction as part of a compositional framework, that handles a more general class of periodic signals. As our approach subsumes temporal decomposition adopted from the framework in [57], we first apply *circuit forwarding* [57] for duration d (i.e., unrolling the reset states of a circuit by d steps) before unfolding is performed.

Figure 2 illustrates the flow of phase abstraction. The process begins by using cube simulation to identify a set of periodic signals as defined in Sect. 3 and computing an optimal duration and phase number based on a selected cube lasso. Once the circuit is unfolded n times, factoring is performed by assigning constant values to the clock-like signals as well as replacing latches with their equivalent or antivalent representative latches in each phase. Next, the property is rewritten by applying structural rewriting techniques and the circuit is further simplified using cone-of-influence reduction. Finally, the simplified circuit (C_{n+4} in Fig. 2) is checked using a base model checking approach such as IC3/PDR [17] or continue to be preprocessed further.

In Fig. 3, we show intuitively an example of a circuit with 4-bit states representing $0, \dots, 9$ and so on, where the initial state is 0. After forwarding the circuit by one step ($d = 1$), the initial state becomes 1. Subsequently with an unfolding of $n = 3$, every state (marked with rectangles) in the unfolded circuit consists of three states from the original circuit. We introduce the formal definitions below.

Unfolding a circuit simply means to copy the transition function multiple times to compute n steps of the original circuit at once. Each copy of the transition function only has to deal with a single phase and can therefore be optimized.

Definition 3 (Unfolded circuit). *Given a circuit $C = (I, L, R, F, P)$ and a phase number $n \in \mathbb{N}^+$. The unfolded circuit $C' = (I', L', R', F', P')$ is:*

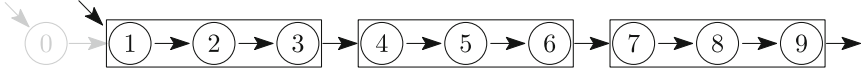


Fig. 3. An example of a forwarded ($d = 1$) and unfolded ($n = 3$) circuit. The circles denote states in the original circuit (0 is the initial state). The rectangles are states in the unfolded circuit.

1. $I' = I^0 \cup \dots \cup I^{n-1}$; $L' = L^0 \cup \dots \cup L^{n-1}$.
2. $R' = \{r'_l \mid l \in L'\}$: for $l \in L^0$, $r'_l = r_l$;
for $i \in (0, n)$, $l^i \in L^i$, $r'_{l^i} = F(I^i, L^{i-1})$.
3. $F' = \{f'_l \mid l \in L'\}$: for $l \in L^0$, $f'_l = f_l(I^0, L^{n-1})$;
for $i \in (0, n)$, $l^i \in L^i$, $f'_{l^i} = f_{l^i}(I^i, L^{i-1})$.
4. $P' = \bigwedge_{i \in [0, n)} P(I^i, L^i)$.

We obtain a simplified circuit by replacing the periodic signals with constants and equivalent/antivalent latches in the unfolded circuit.

Definition 4 (Factor circuit). For a fixed duration d and phase number n , given a d -forwarded and n -unfolded circuit $C = (I, L, R, F, P)$ and a periodic signal with duration d and phase number n for each latch, the factor circuit $C' = (I, L, R', F', P)$ is defined by:

$$\begin{array}{ll}
 R' = \{r'_l \mid l \in L\} : & F' = \{f'_l \mid l \in L\} : \\
 - r'_{l^i} = \lambda_l^i, \text{ if } \lambda_l^i \in \{\perp, \top\}; & - f'_{l^i} = \lambda_l^i, \text{ if } \lambda_l^i \in \{\perp, \top\}; \\
 - r'_{l^i} = r_{\lambda_l^i}, \text{ if } \lambda_l^i \in L. & - f'_{l^i} = f_{\lambda_l^i}, \text{ if } \lambda_l^i \in L. \\
 - r'_{l^i} = \neg r_{\neg \lambda_l^i}, \text{ otherwise.} & - f'_{l^i} = \neg f_{\neg \lambda_l^i}, \text{ otherwise.}
 \end{array}$$

Replaced latches will be removed by a combination of rewriting and cone-of-influence reduction introduced in the following definitions. There are various rewriting techniques also including SAT sweeping [30, 38, 43–45, 59].

Definition 5 (Rewrite circuit). Given a circuit $C = (I, L, R, F, P)$, a rewrite circuit $C' = (I, L, R, F, P')$ satisfies $P \equiv P'$.

For a given circuit, we apply cone-of-influence reduction to obtain a reduced circuit such that latches and inputs outside the cone of influence are removed.

Definition 6 (Reduced circuit). Given a circuit $C = (I, L, R, F, P)$. The reduced circuit $C' = (I', L', R', F', P)$ is defined as follows:

$$\begin{array}{ll}
 - I' = I \cap \text{coi}(P); & - L' = L \cap \text{coi}(P); \\
 - R' = \{r_l \mid l \in L'\}; & - F' = \{f_l \mid l \in L'\},
 \end{array}$$

where the cone of influence of the property $\text{coi}(P) \subseteq (I \cup L)$ is defined as the smallest set of inputs and latches such that $\text{vars}(P) \subseteq \text{coi}(P)$ as well as $\text{vars}(r_l) \subseteq \text{coi}(P)$ and $\text{vars}(f_l) \subseteq \text{coi}(P)$ for all latches $l \in \text{coi}(P)$.

5 Certification

We define a revised certificate format that allows smaller and more optimized certificates. We then propose a method for producing certificates for phase abstraction. The proofs for our main theorems can be found in the Appendix.

5.1 Restricted Simulation

In the following, we define a new variant of the stratified simulation relation [15], which we call *restricted simulation*, that considers the intersection of latches shared between two given circuits as a common component.

Definition 7 (Restricted Simulation). *Given stratified circuits C' and C with $C' = (I', L', R', F', P')$ and $C = (I, L, R, F, P)$. We say C' simulates C under the restricted simulation relation iff*

1. For $l \in (L \cap L')$, $r_l(I, L) \equiv r'_l(I', L')$.
2. For $l \in (L \cap L')$, $f_l(I, L) \equiv f'_l(I', L')$.
3. $P'(I', L') \Rightarrow P(I, L)$.

This new simulation relation differs from [15, 56], where inputs were required to be identical in both circuits ($I = I'$), and latches in C had to form a subset of latches in C' ($L \subseteq L'$). Therefore, under those previous “combinational” [56] or “stratified” [15] simulation relations the simulating circuit C' cannot have fewer latches than L . This is a feature we need for instance when incorporating certificates for cone-of-influence reduction [22], a common preprocessing technique. It opens up the possibility to reduce certificate sizes substantially.

Still, as for stratified simulation, restricted simulation can be verified by three simple SAT checks, i.e., separately for each of the three requirements in Definition 7.

Definition 8 (Semantic independence). *Let \mathcal{V} be a set of variables and $v \in \mathcal{V}$. Then a formula $f(\mathcal{V})$ is said to be semantically independent of v iff*

$$f(\mathcal{V})|_v \equiv f(\mathcal{V})|_{\neg v}.$$

Semantic dependency [28, 40, 49, 53] allows us to assume that a formula only depends on a subset of variables, which without loss of generality simplifies proofs used for the rest of this section. The stratified assumption for reset functions entails no cyclic dependencies thus $R'(L')$ is satisfiable. A reset state in a circuit is simply a satisfying assignment to the reset predicate $R(L)$. Based on the reset condition (Definition 7.1), it is however necessary to show that for every reset state in C it can always be extended to a reset state in C' , while the common variables have the same assignment in both circuits. This is stated in the lemma below, and the proofs can be found in the Appendix.

Lemma 1. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits satisfying the reset condition defined in Definition 7.1. Then $R'(L \cap L')$ is semantically dependent only on their common variables.*

In fact, semantic independence is a direct consequence of restricted simulation; thus no separate check is required. We make a further remark that if the reset function is dependent on an input variable, then it has to be an input variable common to both circuits.

Based on this, we conclude with the main theorem for restricted simulation such that C is safe if C' is safe (i.e., no bad state that violates the property is reachable from any initial state).

Theorem 1. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits, where C' simulates C under restricted simulation. If C' is safe, then C is also safe.*

Intuitively, if there is an unsafe trace in C , Definition 7.1 together with Lemma 1 allow us to find a simulating reset state and transition it with Definition 7.2 to a simulating state also violating the property in C' by Definition 7.3. Here a state in C' simulates a state in C if they match on all common variables. Building on this, we present witness circuits as a format for certificates. Verifying the restricted simulation relation requires three SAT checks, and another three SAT checks are needed for validating the inductive invariant [56]. Therefore certification requires in total six SAT checks as well as a polynomial time check for reset stratification.

Definition 9 (Witness circuit). *Let $C = (I, L, R, F, P)$ be a stratified circuit. A witness circuit $W = (J, M, S, G, Q)$ of C satisfies the following:*

- W simulates C under the restricted simulation relation.
- Q is an inductive invariant in W .

The witness circuit format subsumes [15,57], thus every witness circuit in their format is also valid under Definition 9.

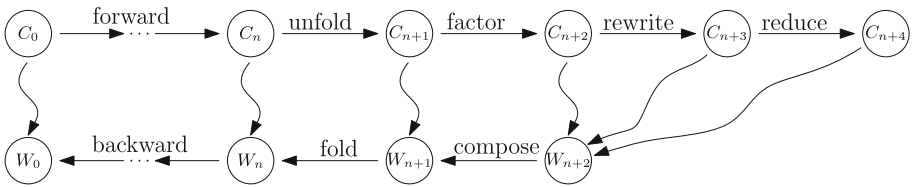


Fig. 4. Certification for (extended) phase abstraction. Base model checking is performed on circuit C_{n+4} , which produces a witness circuit W_{n+2} , that certifies C_{n+2} , C_{n+3} , and C_{n+4} . We construct step-wise to obtain W_0 , which is a certificate for the entire model checking procedure.

5.2 Certifying Phase Abstraction

The certificate format is generic, subsumes [57], and is designed to potentially be used as a standard in future hardware model checking competitions. We proceed

to demonstrate how a certificate can be constructed for a model checking pipeline that includes phase abstraction. The theorems in this section state that this construction guarantees that a certificate will be produced. We illustrate our certification pipeline in Fig. 4. After phase abstraction and base model checking, we can build a certificate backwards based on the certificate produced by the base model checker. The following theorem states that the witness circuit of the reduced circuit serves as a witness circuit for the original circuit too.

Theorem 2. *Given a circuit $C = (I, L, R, F, P)$ and its reduced circuit $C' = (I', L', R', F', P')$. A witness circuit of C' is also a witness circuit of C .*

The outcome of rewriting is a circuit with a simplified property that maintains semantic equivalence with the original property. Therefore in our framework, the certificate for the simplified property is also valid for the original property. Furthermore, certificates can be optimized by rewriting at any stage. We summarize this in the following proposition.

Proposition 1. *Given a circuit C and its rewrite circuit C' . A witness circuit of C' is also a witness circuit of C .*

We define the composite witness circuit to combine the certificates for cube simulation and the factor circuit.

Definition 10 (Composite witness circuit). *Given a stratified circuit $C = (I, L, R, F, P)$ and its factor circuit $C' = (I', L', R', F', P')$, and the unfolded loop invariant $\phi = \bigvee_{i \in [0, m]} \bigwedge_{j \in [0, n]} c_{i * n + j + d}$, with $m = (\delta + \omega - d + 1) / n$, obtained from the cube lasso. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C' . The composite witness circuit $W = (J, M, S, G, Q)$ is defined as follows:*

1. $J = I \cup J'$.
2. $M = L \cup (M' \setminus L')$.
3. $S = \{s_l \mid l \in M\}$:
 - (a) for $l \in L, s_l = r_l$;
 - (b) for $l \in M' \setminus L', s_l = s'_l$.
4. $G = \{g_l \mid l \in M\}$:
 - (a) for $l \in L, g_l = f_l$;
 - (b) for $l \in M' \setminus L', g_l = g'_l$.
5. $Q = \phi(L) \wedge Q'(J', M')$.

Theorem 3. *Given circuit $C = (I, L, R, F, P)$, and factor circuit $C' = (I', L', R', F', P')$. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C' , and $W = (J, M, S, G, Q)$ constructed as in Definition 10. Then W is a witness circuit of C .*

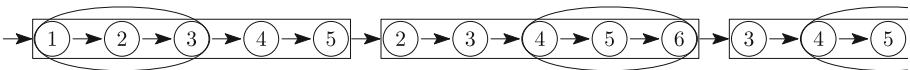


Fig. 5. Every fully initialized state of a 3-folded witness circuit contains 3 original states that form an unfolded state. Two consecutive 3-folded states contain either the same unfolded states or two states consecutive in the unfolded circuit.

In the construction of an n -folded witness circuit from the unfolded witness W' , a single instance of W' 's latches (N), yet multiples of the original latches L are used. As illustrated in Fig. 5, these L record a history, contrasting with their role in the unfolded circuit where they calculate multi-step transitions.

Definition 11 (n -folded witness circuit). *Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, and its unfolded circuit $C' = (I', L', R', F', P')$. Let $W' = (J', M', S', G', Q')$ be the witness circuit of C' . The n -folded witness circuit $W = (J, M, S, G, Q)$ is defined as follows:*

1. $J = I^0 \cup J^0$, where I^0 and J^0 are I and J' respectively.
2. $M = I^1 \dots I^m \cup L^0 \dots L^m \cup N \cup J^1 \cup \{b^0 \dots b^m, e^0 \dots e^{n-2}\}$,
where $m = 2 \times n - 2$, $N = M' \setminus L'$, and I^i, L^i are copies of I and L , and J^1 is a copy of J' .
3. $S = \{s_l \mid l \in M\}$:
 - (a) $s_{b^0} = \top$;
 - (b) For $i \in (0, m]$, $s_{b^i} = \perp$.
 - (c) For $i \in [0, n - 1]$, $s_{e^i} = \perp$.
 - (d) For $l \in L^0$, $s_l = r'_l$.
 - (e) For $l \in (I^1 \dots I^m \cup L^1 \dots L^m \cup J^1)$, $s_l = l$.
 - (f) For $l \in N$, $s_l = s'_l$.
4. $G = \{g_l \mid l \in M\}$:
 - (a) $g_{b^0} = \top$.
 - (b) For $i \in [1, m]$, $g_{b^i} = b^{i-1}$.
 - (c) $g_{e^0} = b^{n-1} \wedge \neg e^{n-2}$.
 - (d) For $i \in [1, n - 1]$, $g_{e^i} = e^{i-1} \wedge \neg e^{n-2}$.
 - (e) For $l \in L^0$, $g_l = f_l$.
 - (f) For $l^1 \in J^1$, $g_{l^1} = l^0$.
 - (g) For $i \in [1, m]$, $l^i \in (I^i \cup L^i)$, $g_{l^i} = l^{i-1}$.
 - (h) For $l \in N$,
 $g_l = ite(e^{n-2}, g'_l(J^1, M' \cap (I^{m-n+1} \dots I^m \dots L^{m-n+1} \dots L^m \cup N)), l)$.
5. $Q = \bigwedge_{i \in [0, 6]} q^i$:
 - (a) $q^0 = P(I^0, L^0)$.
 - (b) $q^1 = b^0$.
 - (c) $q^2 = \bigwedge_{i \in [1, m]} (b^i \rightarrow b^{i-1})$.
 - (d) $q^3 = \bigwedge_{i \in [1, m]} (b^i \rightarrow (L^i \simeq F(I^{i-1}, L^{i-1})))$.
 - (e) $q^4 = \bigwedge_{i \in [1, m]} ((\neg b^i \wedge b^{i-1}) \rightarrow (R(L^{i-1}) \wedge S'(N)))$.
 - (f) $q^5 = b^m \rightarrow (\bigvee_{i \in [0, n]} ((\bigwedge_{j \in [i, n-1]} \neg e^j) \wedge (\bigwedge_{j \in [0, i]} e^j) \wedge Q'(J^0, M' \cap (L^i \dots \cup L^{i+n-1} \cup N))))$.
 - (g) $q^6 = \bigwedge_{i \in [1, n-2]} (e^i \rightarrow e^{i-1})$.
 - (h) $q^7 = \bigwedge_{i \in [0, n-2]} (e^i \rightarrow b^{n+i})$.

$$(i) \quad q^8 = \bigwedge_{i \in [0, n-2]} ((\neg b^m \wedge b^{n+i}) \rightarrow e^i).$$

The b^i s are used for encoding initialization. So that inductiveness is ensured when not all copies are initialized. The $n - 1$ bits e^i are used to determine which set of n consecutive original states form an unfolded state (a state in the unfolded circuit). This information is used to determine on which copies the unfolded property needs to hold and to transition the latches in N (the part of the witness circuit added by the backend model checker) once every n steps.

Theorem 4. *Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, its unfolded circuit $C' = (I', L', R', F', P')$ with a witness circuit $W' = (J', M', S', G', Q')$. Let $W = (J, M, S, G, Q)$ be the circuit constructed as in Definition 11. Then W is a witness circuit of C .*

After the witness circuit has been folded, the same construction from [57] can be used to construct the backward witness. With that, the pipeline outlined in Fig. 4 is completed. If phase abstraction is the first technique applied by the model checker, a final witness is obtained. Otherwise, further witness processing steps still need to be performed. An example of the entire process is illustrated in Fig. 6.

6 Implementation

In this section, we present MC2, a certifying model checker implementing phase abstraction and IC3. We implement our own IC3 since no existing model checker supports reset functions or produces certificates in the desired format. We used fuzzing to increase trust in our tool. The version of MC2 used for the evaluation, was tested on over 25 million randomly generated circuits [14] in combination with random parameter configurations. All produced certificates were verified.

To extract periodic signals we perform ternary simulation [52] while using a forward-subsumption algorithm based on a one-watch-literal data structure [58] to identify supersets of previously visited cubes, and thereby a set of cube lassos. For each cube lasso we consider every factor of the loop length ω as a phase number candidate n . We also consider every duration d , that renders the leftover tail length $(\delta - d)$ divisible by n . To keep the circuit sizes manageable, we limit both n and d to a maximum of 8. We call each pair (d, n) an unfolding candidate and compute the corresponding periodic signal (Definition 2) for each latch.

For each phase, equivalences are identified by inserting a bit string corresponding to the signs of each latch into a hash table. After identifying the signals, forwarding and unfolding are performed on a copy of the circuit, followed by rudimentary rewriting. Currently the rewriting does not include structural hashing and is mostly limited to constant propagation. Afterwards a sequential cone-of-influence analysis starting from the property is performed. After performing these steps for each candidate, we pick the duration-phase pair that yields a circuit with the fewest latches and give it to a backend model checker.

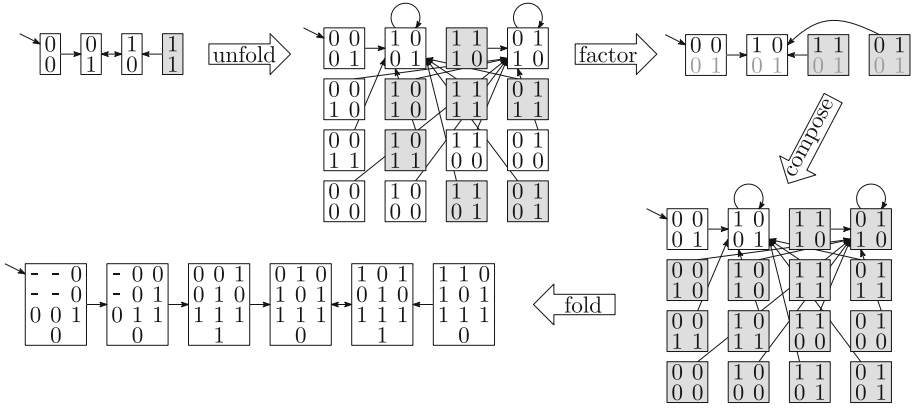


Fig. 6. A concrete example of the model checking and certification pipeline. The original circuit has two latches; the bottom latch alternates and the top copies the previous value of this clock. The property is that at least one bit is unset. Bad states are marked gray. After unfolding with phase number two, the size of the state space is squared. Since the bottom bit is periodic, we can replace it with a constant in each phase (factor). On this circuit terminal model checking is performed, since the property is already inductive (no transition from good to bad), the circuit serves as its own witness. To produce the final witness circuit, the clock is added back as a latch, and the property is extended with the loop invariant asserting that the clock has the correct value for each phase. Lastly, the circuit is *folded* to match the speed of the original circuit. Three initialization bits b^i are introduced and one additional bit e^0 that determines which pair of consecutive states need to fulfill the property (0 for the right pair and 1 for the left). This check is only part of the property once full initialization is reached. For this final witness circuit, only the good states are depicted. Also, the first two states represent sets of good states with the same behavior.

We evaluated the preprocessor on three backend model checkers: the open-source k -induction-based model checker McAiger [12] (Kind in the following), the state-of-the-art IC3 implementation in ABC [18] and our own version of IC3 that supports reset functions and produces certificates. Since ABC does not support reset functions, it is not able to model check any forwarded circuit (note that implementing this feature on ABC is also a non-trivial task), therefore for this configuration we only ran phase abstraction without forwarding thus no temporal decomposition.

Our IC3 implementation on MC2 does feature cube minimization via ternary simulation [25], however it is missing proof-obligation rescheduling. In fact, we currently use a simple stack of proof obligations as opposed to a priority queue. Despite using one SAT solver instance per frame, we also do not feature cones-on-demand, but instead always translate the entire circuit using Tseitin [55].

Lastly, we also modified the open source implementation of Certifaiger [21] to support certificates based on restricted simulation. For a witness circuit C' of

C , the new certificate checker encodes the following six checks as combinatorial AIGER circuits and then uses the `aigtoconf` to translate them to SAT:

- Ⓐ The property of C' holds in all initial states.
- Ⓑ The property of C' implies the property for successor states.
- Ⓒ The property of C' holds in all good states.
- Ⓓ The reset functions of common latches are equivalent. (Definition 7.1)
- Ⓔ The transition functions of common latches are equivalent. (Definition 7.2)
- Ⓕ The property of C' implies the property of C . (Definition 7.3)

The first three checks are unchanged and encode the standard check for P' being an inductive invariant in C' . Since P' is both the inductive invariant and the property we are checking, Ⓒ can technically be omitted. However, in our implementation, the inductiveness checker is an independent component from the simulation checker and would also work for scenarios where the inductive invariant is a strengthening of the property in C' .

7 Experimental Evaluation

This section presents experimental results for evaluating the impact of preprocessing on the different backends, as well as the effectiveness of our proposed certification approach. The experiments were run in parallel on a cluster of 32 nodes. Each node was equipped with two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz and 128 GB of main memory. We allocated 8 instances to each node, with a timeout of 5000 s for model checking and 10 000 s for certificate checking. Memory is limited to 8 GB per instance in both cases.

The benchmarks are obtained from HWMCC2010 [13] which contains a good number of industrial problems. As we observe from the experiments in general, preprocessing is usually fast. Ignoring one outlier in our benchmark set, it completes within an average of 0.07 seconds and evaluates no more than 17 unfolding candidates per benchmark. Interestingly, for the outlier “bobsmnut1”, 3019 unfoldings are computed for 179 different cube lassos within 34 seconds.

Table 1 presents the effect of our preprocessing on different backends, further illustrated in Fig. 7. Our preprocessor was able to improve the performance of the sophisticated IC3/PDR implementation in ABC, allowing us to solve five more instances, all from the intel family. For each benchmark from this family, our heuristic computed an optimal phase number of 2. A likely explanation for this is that the real-world industrial designs tend to contain strict two-phase clocks [6]. The positive effect of phase abstraction is also clear in combination with the k -induction (Kind) backend. Circuit forwarding provides a further improvement, that is especially notable on the prodcell benchmarks. These also illustrate how forwarding enables more successful unfolding. Without forwarding, preprocessing only unfolds 61 out of the 818 benchmarks with an average phase number of 2, with forwarding 152 circuits are unfolded with an average of 4.

Even though our prototype implementation of IC3 is missing a number of important features present in ABC, it still solves a large number of benchmarks.

Table 1. We presents the effect of preprocessing in combination with different backend engines on model checking time. We compare no preprocessing to only phase abstraction without forwarding (PA) and full preprocessing (Full). Note that, ABC does not support reset functions and can therefore not be combined with full preprocessing. For each model we present the phase number without forwarding \bar{n} for PA and the duration d and phase number n corresponding to Full. Models where the property holds are marked as safe. The first two rows present the number of solved instances and the PAR2 score [29] over all 818 benchmarks. The table shows all instances where preprocessing had either a positive or negative impact on model checking success, with the exception of those instances rendered unsolvable for our IC3 implementation by forwarding.

	Model				ABC		Our IC3			Kind		
	Safe	\bar{n}	d	n		PA	PA	Full		PA	Full	
Solved					740	745	715	715	604	533	538	544
PAR2					996	941	1357	1351	2699	3533	3472	3399
abp4p2ff	✓	1	1	1	1.12	1.08	6.35	6.23	6.18	2.50	2.50	
bjrb07		3	0	3	0.12	0.10	0.11	0.03	0.07		0.03	0.04
nusmvb5p2		5	0	5	0.12	0.10	0.01	0.01	0.01		0.01	0.01
nusmvb10p2		5	0	5	0.22	0.13	0.10	0.03	0.04		0.02	0.02
prodcell0	✓	1	5	8	26.97	27.07	228.46	243.73	49.76			2.37
prodcell0neg	✓	1	5	8	16.36	15.93	230.57	230.67	36.62			2.39
prodcell1	✓	1	7	8	23.45	23.38	654.21	665.86	59.67			4.43
prodcell1neg	✓	1	7	8	28.36	28.33	681.11	738.61	61.74			4.48
prodcell2	✓	1	7	8	24.98	24.58	661.71	663.37	56.74			4.43
prodcell2neg	✓	1	7	8	20.23	20.28	778.39	768.75	56.14			4.47
bc57sen0neg	✓	1	1	1	503.61	494.55	910.72	906.87	1760.41			830.92
abp4ptimo	✓	1	1	1	4.14	4.13	28.93	29.91	6.32			608.55
boblivea		1	2	1	3.70	3.68			7.85			
bobsm5378d2		1	8	1	4.04	4.12			88.36			
bobsmnut1		8	5	8	10.95	40.08			2504.07			
prodcell3neg	✓	2	2	8	27.88	10.86	310.22				837.43	2.73
prodcell4neg	✓	2	2	8	44.31	9.90	404.12				26.04	2.77
prodcell3	✓	2	2	8	23.45	11.24	320.23			1103.29	19.22	2.48
prodcell4	✓	2	2	8	31.40	10.08	398.83			29.71	18.67	2.68
pdtvisvsar29		1	2	5					1523.73	0.36	0.29	0.40
intel042	✓	1	3	2						3876.04	4061.38	
intel022		2	2	2		1852.29						
intel021		2	2	2		2752.86		651.56				
intel023		2	2	2		2257.94		3728.38				
intel029		2	2	2		2550.14		3437.64				
intel024		2	2	2		167.96	676.64	4526.60				
intel019		2	2	2				2716.40				

However, as opposed to ABC it does lose a number of benchmarks with phase abstraction. This can be explained by the lack of sophisticated rewriting that can exploit the unfolded circuits structure. The addition of forwarding is highly

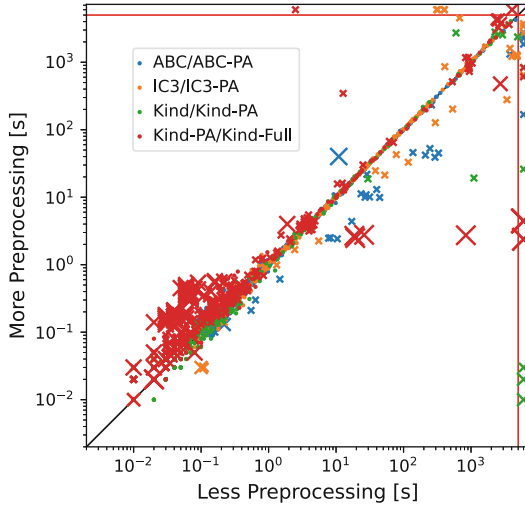


Fig. 7. Comparison of model checking performance. We compare four pairs of configurations; the three backend engines with and without phase abstraction (with fixed duration 0) and for Kind we present the effect of additionally allowing forwarding. The size of the markers represents $n + d$. The dots represent instances where the preprocessing heuristic decided not to alter the circuit. The red lines mark the timeout of 5000s. Markers beyond that line represent instances solved by one configuration but not the other. (Color figure online)

detrimental to performance, losing 115 instances. This is due to our implementation following the PDR design outlined in [25]. It requires any blocked cube not to intersect the initial states after generalization. If only a single reset state exists this check is linear in the size of the cube. However, in the presence of reset functions it is implemented with a SAT call. While also slower the main problem however is that the reset-intersection check is also more likely to block generalization. On the 115 lost benchmarks generalization failed 96% of the time, while it only failed in 1.8% of the cases without forwarding. We keep the optimization of our IC3 implementation in the presence of reset functions for future work.

Figure 8 displays certification results on MC2 in comparison to model checking time. IC3 provides certificates that are easily verifiable, as confirmed by our experiments with cumulative overhead of only 3%. The addition of phase abstraction (i.e., including constructing n -folded witnesses as in Fig. 4, without witness back-warding) does not bring significant additional overhead. When forwarding is allowed, the certification overhead increases to 10%. The run time of certificates generation and encoding to SAT is negligible for all configurations. The certification time is dominated by the SAT solving time for the transition (Definition 7.2) and consecution check. Overall, this is a significant improvement over related work from [57] which reported 1154% overhead on the same set of benchmarks using a k -induction engine as the backend.

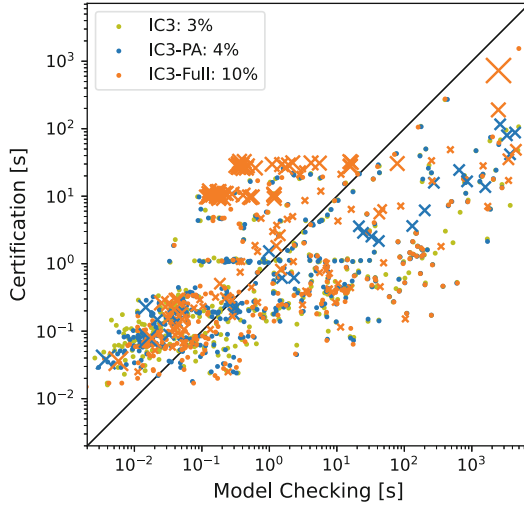


Fig. 8. Certification vs. model checking time for three configurations of our IC3 engine. The legend shows the cumulative overhead of including certification for all solved instances. The size of the markers represents $n + d$. The dots represent instances where preprocessing did not alter the circuit.

8 Conclusion

In this paper, we present a certificate format that can be effectively validated by an independent certificate checker. We demonstrate its versatility by applying it to an extended version of phase abstraction, which we introduce as one of the contributions of this paper. We have implemented the proposed approach on a new certifying model checker MC2. The experimental results on HWMCC instances show that our approach is effective and yields very small certification overhead, as a vast improvement over related work. Our certificate format allows for smaller certificates and is designed to be possibly used in hardware model checking competitions as a standardized format.

Beyond increasing trust in model checking, certificates can be utilized in many other scenarios. For instance, such certificates will allow the use of model checkers as additional hammers in interactive theorem provers such as Isabelle [48] via Sledgehammer [50], with the potential of significantly reducing the effort needed for using theorem provers in domains where model checking is essential, such as formal hardware verification, our main application of interest. Currently in Isabelle, Sledgehammer allows to encode the current goal for automatic theorem provers or SMT solvers and then call one of many tools to solve the problem. The tool then provides a certificate which is lifted to a proof that can be replayed in Isabelle. We plan to add our model checker as an additional hammer to increase the automatic proof capability of Isabelle. This further motivates us to investigate certificate trimming via SAT proofs.

Acknowledgements. This work is supported by the Austrian Science Fund (FWF) under the project W1255-N23, the LIT AI Lab funded by the State of Upper Austria, the ERC-2020-AdG 101020093, the Academy of Finland under the project 336092 and by a gift from Intel Corporation.

References

1. Amjad, H.: Programming a symbolic model checker in a fully expansive theorem prover. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 171–187. Springer, Heidelberg (2003). https://doi.org/10.1007/10930755_11
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Balyo, T., Heule, M.J.H.: Proceedings of SAT competition 2016 – solver and benchmark descriptions. Department of Computer Science Series of Publications B, vol. B-2016-1. University of Helsinki (2016)
4. Barbosa, H., et al.: Generating and exploiting automated reasoning proof certificates. *Commun. ACM* **66**(10), 86–95 (2023). <https://doi.org/10.1145/3587692>
5. Barbosa, H., et al.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 15–35. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_3
6. Baumgartner, J., Heyman, T., Singhal, V., Aziz, A.: Model checking the IBM gigahertz processor: an abstraction algorithm for high-performance netlists. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 72–83. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_9
7. Baumgartner, J., Heyman, T., Singhal, V., Aziz, A.: An abstraction algorithm for the verification of level-sensitive latch-based netlists. *Formal Methods Syst. Des.* **23**, 39–65 (2003)
8. Beyer, D., Chien, P., Lee, N.: Bridging hardware and software analysis with Btor2C: a word-level-circuit-to-C translator. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13994, pp. 152–172. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_12
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: SIGSOFT FSE, pp. 326–337. ACM (2016)
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. *ACM Trans. Softw. Eng. Methodol.* **31**(4), 57:1–57:69 (2022)
11. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 739–764. IOS Press (2021). <https://doi.org/10.3233/FAIA201002>
12. Biere, A., Brummayer, R.: Consistency checking of all different constraints over bit-vectors within a SAT solver. In: FMCAD, pp. 1–4. IEEE (2008)
13. Biere, A., Claessen, K.: Hardware model checking competition 2010 (2010). <http://fmv.jku.at/hwmcc10/>
14. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report, FMV Reports Series, Inst. FMV, JKU Linz, Austria (2011)
15. Biere, A., Yu, E., Frolejks, N.: Stratified certification for k-induction. In: FMCAD, vol. 3, p. 59. TU Wien Academic Press (2022)
16. Bjesse, P., Kukula, J.H.: Automatic generalized phase abstraction for formal verification. In: ICCAD, pp. 1076–1082. IEEE Computer Society (2005)

17. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
18. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
19. Case, M.L., Baumgartner, J., Mony, H., Kanzelman, R.: Approximate reachability with combined symbolic and ternary simulation. In: FMCAD, pp. 109–115. FMCAD Inc. (2011)
20. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: FMCAD, pp. 17–24. IEEE (2009)
21. Certifaiger: Certifaiger (2021). <http://fmv.jku.at/certifaiger>
22. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model Checking, 2nd edn. MIT Press, Cambridge (2018)
23. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-10575-8>
24. Degtyarev, A., Voronkov, A.: Equality reasoning in sequent-based calculi. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 611–706. Elsevier and MIT Press (2001)
25. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134. FMCAD Inc. (2011)
26. van Eijk, C.A.J., Jess, J.A.G.: Exploiting functional dependencies in finite state machine verification. In: ED&TC, pp. 9–14. IEEE Computer Society (1996)
27. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
28. Fleury, M., Biere, A.: Mining definitions in Kissat with Kittens. Formal Methods Syst. Des. 1–24 (2023)
29. Frolleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: Sat competition 2020. Artif. Intell. **301**, 103572 (2021)
30. Fujita, M.: Toward unification of synthesis and verification in topologically constrained logic design. Proc. IEEE **103**(11), 2052–2060 (2015)
31. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: FMCAD, pp. 1–9. IEEE (2018)
32. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for SAT-based model checking. Formal Methods Syst. Des. **57**(2), 178–210 (2021)
33. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
34. Heule, M.J.: Proofs of unsatisfiability. In: Handbook of Satisfiability, pp. 635–668. IOS Press (2021)
35. Heule, M.J., Biere, A.: Proofs for satisfiability problems. In: All About Proofs, Proofs for All, vol. 55, no. 1, pp. 1–22 (2015)
36. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, 11–12 August 2022. CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022)

37. Kaufmann, D., Fleury, M., Biere, A., Kauers, M.: Practical algebraic calculus and nullstellensatz with the checkers pacheck and pastèque and nuss-checker. *Formal Methods Syst. Des.* 1–35 (2022)
38. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **21**(12), 1377–1394 (2002)
39. Kuismin, T., Heljanko, K.: Increasing confidence in liveness model checking results with proofs. In: Bertacco, V., Legay, A. (eds.) *HVC 2013*. LNCS, vol. 8244, pp. 32–43. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_3
40. Lagniez, J.M., Lonca, E., Marquis, P.: Definability for model counting. *Artif. Intell.* **281**, 103229 (2020)
41. Lammich, P.: Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.* **64**(3), 513–532 (2020)
42. Mebsout, A., Tinelli, C.: Proof certificates for SMT-based model checkers for infinite-state systems. In: *FMCAD*, pp. 117–124. IEEE (2016)
43. Mishchenko, A., Chatterjee, S., Brayton, R.K.: Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In: *DAC*, pp. 532–535. ACM (2006)
44. Mishchenko, A., Chatterjee, S., Brayton, R.K., Eén, N.: Improvements to combinational equivalence checking. In: *ICCAD*, pp. 836–843. ACM (2006)
45. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: a unifying representation for logic synthesis and verification. Technical report, ERL Technical Report (2005)
46. Mony, H., Baumgartner, J., Aziz, A.: Exploiting constraints in transformation-based verification. In: Borrione, D., Paul, W. (eds.) *CHARME 2005*. LNCS, vol. 3725, pp. 269–284. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548_21
47. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) *CAV 2001*. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2
48. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
49. Padoa, A.: Essai d’une théorie algébrique des nombres entiers, précédé d’une introduction logique à une théorie déductive quelconque. In: *Bibliothèque du Congrès international de philosophie*, vol. 3, pp. 309–365 (1901)
50. Paulson, L., Nipkow, T.: The sledgehammer: let automatic theorem provers write your isabelle scripts (2023)
51. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021*, Pittsburg, PA, USA, 11 July 2021. *EPTCS*, vol. 336, pp. 49–54 (2021). <https://doi.org/10.4204/EPTCS.336.6>
52. Seger, C.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods Syst. Des.* **6**(2), 147–189 (1995)
53. Slivovsky, F.: Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In: Lahiri, S.K., Wang, C. (eds.) *CAV 2020*. LNCS, vol. 12224, pp. 508–528. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_24
54. Sprenger, C.: A verified model checker for the modal μ -calculus in Coq. In: Steffen, B. (ed.) *TACAS 1998*. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054171>

55. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pp. 466–483 (1983)
56. Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model checking results. In: Silva, A., Leino, K.R.M. (eds.) *CAV 2021*. LNCS, vol. 12760, pp. 363–386. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_17
57. Yu, E., Froleyks, N., Biere, A., Heljanko, K.: Towards compositional hardware model checking certification. In: *FMCAD (2023)*
58. Zhang, L.: On subsumption removal and on-the-fly CNF simplification. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005*. LNCS, vol. 3569, pp. 482–489. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_42
59. Zhu, Q., Kitchen, N., Kuehlmann, A., Sangiovanni-Vincentelli, A.L.: SAT sweeping with local observability don't-cares. In: *DAC*, pp. 229–234. ACM (2006)

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

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





Verifying a Realistic Mutable Hash Table Case Study (Short Paper)

Samuel Chassot^(✉)  and Viktor Kunčák 

EPFL, Lausanne, Switzerland

{samuel.chassot,viktor.kuncak}@epfl.ch

Abstract. In this work, we verify, using the Stainless program verifier, the mutable `LongMap` from the Scala standard library, a hash table using open addressing within a single array. As an executable specification, we write an immutable map based on a list of tuples and verify it against the mathematical definition of a map. We then show that `LongMap`'s operations correspond to operations of this association list. To express the resizing of the hash table array, we introduce a new reference-swapping construct in Stainless. This allows us to apply the decorator design pattern without introducing aliasing. Our verification effort led us to find and fix a bug in the original implementation that manifests for large hash tables. Our performance analysis shows the verified version to be within a 1.5 factor of the original data structure.

Keywords: Formal verification · Hash table · LongMap · Scala

1 Introduction

With the improvements in effectiveness and expanding user base of proof assistants such as Isabelle/HOL [22] and Coq [27], we are witnessing systematic verification of many purely functional data structures. The verification of these data structures is highly effective using those tools. In the Scala language ecosystem, such verification efforts were carried out using the Stainless verifier [13] and its predecessor Leon [19]. However, verification of mutable data structures remains more challenging. As an example for hash table validation on the JVM platform, a recent attempt [8] provided a proof with interactive steps and an incomplete proof based on bounded model checking for one function. We consider such efforts very valuable. At the same time, our verification led us to discover a bug that bounded model checking would have likely missed due to the large arrays required. This illustrates the limitations of bounded checks and the need for complete formal verification.

In this work, we verify a data structure from the Scala standard library: the mutable `LongMap[V]`, a hash table with keys of type `Long` and values of a generic type `V`, implemented with open addressing (with all data stored in the arrays). We verify it using Stainless, a verification framework for a subset of Scala. To our knowledge, this is the first verified mutable map in Scala and the first verified

hash table with open addressing and non-linear probing. Our implementation closely follows the existing implementation of the Scala library [26], which was implemented with efficiency in mind and withstood the test of usability. This is the fastest hash table implementation we know of in the Scala ecosystem. Our experience helped us further assess the use of Stainless for imperative code, following recent verification of the QOI compression algorithm [5] and file system components [12]. Our paper includes the following contributions:

1. As the key result, the adaptation and full formal verification of the mutable `LongMap` of the Scala standard library [26] using Stainless [13, 20]; this hash table can serve as a basis for other verified project; our code and the SMT queries generated during verification are available on Zenodo [6];
2. A reference implementation of a map verified against the mathematical definition of a map and lemmas for reasoning about such maps. This map is realized as a sorted list of tuples. We use it as an executable specification for `LongMap` and find that it supports automated and inductive reasoning better than the built-in maps of Stainless;
3. Introduction into Stainless of an operator for swapping references, which increases the expressive power of Stainless while preserving non-aliasing, allowing us to implement the resizing and balancing of the hash table;
4. An evaluation of the performance of both `LongMap` implementations (original and verified) and the mutable `HashMap` of the Scala standard library (unverified), showing that the performance of the verified implementation remains competitive despite the changes introduced to simplify verification.

1.1 Related Work

Hash tables have been of interest in verification from the early days of the field. Guttag [11] explores the use of algebraic specifications for reasoning about hash tables, though without formal connection to executable implementations. A hash table is one of the case studies [17] in the Jahob verification system [18, 29]. The version in Jahob does not use open addressing but separate chaining with linked list buckets. Furthermore, that case study uses, as an unverified assumption, the fact that the hash function is pure (total, without side effects, terminating, and deterministic). The Eiffel2 project offers a collection of verified data containers, impressive by its diversity [23]. They implemented and verified a hash table implementation using chaining. These containers are, however, simpler in their implementations than what appears in Scala and Java standard libraries. We could not explore this collection in more depth because the tools used are unavailable. De Boer et al. verified JDK’s `IdentityHashMap`, based on open addressing and *linear* probing, in their case study [8]. The verification was done using KeY [1] and JJBMC [4], both accepting JML specification. They notably did not manage to provide a deductive proof for the `remove` method and one of its auxiliary methods, but instead used bounded model checking for a map of up to four elements. The KeY deductive proofs required interactive steps for the more complex methods, up to 1’655 for the `put` method. Hance et al. also proposed techniques to verify distributed systems interacting with an asynchronous

environment, in particular file systems [14]. In this work, they developed and verified a hash table with open addressing and *linear* probing in Dafny. They implemented two versions of the hash table, one immutable and one mutable. This separates the functional correctness and correct heap manipulation proofs but requires implementing the hash table twice.

2 LongMap: From Scala Library to Stainless

A `LongMap[V]` (called `LongMap` in this work) is a data structure implementing a *map* behavior with keys of type `Long` (signed 64-bit machine integers) and values of generic type `V`. The mutable `LongMap` of the Scala standard library [26] is a hash table employing open addressing and non-linear probing.

We implement a subset of the original `LongMap` interface (outlined in Sect. 3). This subset corresponds to the functions implementing the map functionality (we omit functions specific to the Scala collections hierarchy). The `apply` function returns the value stored for a given key or a default value if absent. The `remove`, `update` (to add/update pairs), and `repack` (to resize/balance the map) functions return a Boolean value indicating the operation success.

The keys and values are stored in two arrays called `_keys` and `_values` respectively. Both are of size $N = 2^n$ for some $3 \leq n \leq 30$. The index of a given key is computed using a hash function. The corresponding value is stored in the second array at the same index as its key. We define `mask = N - 1`.

There are 2 special values in `_keys`: `0` and `Long.MinValue`. The value `0` indicates a free spot while `Long.MinValue` is a *tombstone* value, indicating that a key was removed at this spot.

We use open addressing with non-linear probing to resolve collisions. Following the original Scala implementation [7, 26], the probing function is $index_{x+1} = (index_x + 2 * (x + 1) * x - 3) \& mask$, resulting in cubic index growth. Our verification is independent of the particular probing function but checks that the implementation is pure (i.e., deterministic, total, terminating, and without side effects), free of runtime errors, and returns an index within range.

All operations rely on two elementary ones: 1) looking for a key (`seekEntry`), and 2) looking for a key or an empty spot (`seekEntryOrOpen`). These two operations use non-linear probing, with the special values `0` and `Long.MinValue` in `_keys`. As an example, `update(k: Long, v: V)` starts out by computing $i = \text{seekEntryOrOpen}(k)$. If k is at index i , it writes `_values(i) = v`; if the function returns an open spot, it writes `_keys(i) = k` and `_values(i) = v`.

2.1 Adapting for Verification

Next, we present the changes we made to the original code to comply with the supported subset of Scala, improve the SMT solver performance, make writing specifications easier, and simplify termination checking.

Tail recursion (to ease verification). We replace `while` loops with tail-recursive functions. Stainless can perform this transformation internally, but we

have better control over specifications if we manually transform the source. For example, using a loop invariant makes having different pre- and post-conditions impossible. The Scala compiler transforms tail-recursive functions back to loops during compilation, so no performance is lost.

Loop counters (to prove termination). We introduce a counter and a condition that stops `while` loops (implemented as tail recursion) in `seekEntry` and `seekEntryOrOpen` after a fixed number of iterations. We need this counter as we do not know whether this probing function covers the space of all indices. Moreover, it allows the proof to be agnostic to the probing function. It has a negligible impact on performance, as shown in Sect. 4.

Data representation (for SMT performance). The original implementation uses the MSBs (Most Significant Bits) of the index returned by the seeking functions to indicate whether the index points to the key, a 0, or a tombstone. We replace this with some ADT for better code readability and improved verification experience, as bitwise operations are often slow in SMT solvers.

Typing and initialization of arrays (to comply with subset). In the original implementation, the array storing values (`_values`) is an `Array[AnyRef]`, containing `null` by default, and using casts to store and access values. In our verified implementation, `_values` contains boxed values because Stainless does not support SPSVERBc48s, and the `Array.fill` function (used to instantiate new arrays) does not support generically typed arrays. The boxing is implemented using case classes (i.e., ADTs).

Refactoring (to ease verification). We split the implementation into two classes, following the decorator design pattern (DP), as detailed in Sect. 3.1.

3 Specification and Verification

We first implement `ListLongMap`, an immutable map backed by a strictly ordered list of pairs (`Long, V`), and verify it against the mathematical specification of a map. It serves as the executable specification of the mutable `LongMap`. We thus specify the mutable `LongMap` as behaving as the corresponding `ListLongMap`. A ghost method `map()` (not executed at runtime) of `LongMap` returns an instance of `ListLongMap` with the same content and is used in contracts. For example, `update` is specified as follows: `old(this).map() + (k, v) == this.map()`. Figure 1 shows the `LongMap` interface and specification. Postconditions, expressed using `ensuring` calls, are lambda functions taking the return value as parameter (i.e., `res`). The method `valid` is the data structure representation invariant stating, among other things, that the inserted elements can be found when searching subsequently using the same probing function. Table 1 shows the lines of code for the program, specification, and proofs for both maps.

3.1 Decorator Design Pattern for Modular Proofs

Following the decorator DP, we split the `LongMap` implementation into two classes to better modularize the proof. First, `LongMapFixedSize` implements

```

def contains(key: Long): Boolean = { ...
} ensuring (res => valid && (res == map.contains(key)))
def apply(key: Long): V = { ...
} ensuring (res => valid &&
            (if (contains(key)) res == map.get(key).get
             else res == underlying.v.defaultEntry(key)))
def update(key: Long, v: V): Boolean = { ...
} ensuring (res => valid &&
            (if (res) map.contains(key) && (map == old(this).map + (key, v))
             else map == old(this).map))
def remove(key: Long): Boolean = { ...
} ensuring (res => valid && (if (res) map == old(this).map - key
                           else map == old(this).map))
def repack(): Boolean = { ... } ensuring (res => !res || map == old(this).map)

```

Fig. 1. Mutable LongMap interface and specification (note that we omit preconditions in this figure which are only checks of the invariant (*valid*), if any).

the LongMap specification depicted in Fig. 1 without resizing (with arrays of a given fixed length). Then, we implement the LongMap class as a decorator of LongMapFixedSize. It implements the same interface and adds the resizing operation (**repack** function). Being a decorator, this class forwards all operations to an underlying instance of LongMapFixedSize except for the **repack** function. A key observation about the original implementation of **repack** is that it works very similarly to the **update** function to insert all pairs. Only some checks are omitted because the array is assumed to be fresh (containing no tombstone values and, initially, no keys). This observation allows us to use **update** to implement **repack** without significantly impacting the performance, while simplifying the proof.

3.2 Swap Operation for More Expressive Unique Reference

As discussed in Sect. 3.1, the LongMap class relies on an underlying instance of the LongMapFixedSize class. The underlying instance must be replaced by a new one during calls to **repack**. The repacking process first computes the new array size, then creates a new instance of LongMapFixedSize with this size, inserts all pairs, and finally replaces the current underlying instance with this new one. Aliasing appears during this replacement, yet Stainless disallows it. We can, however, observe that there is no need for aliasing because the reference to the newly created instance is not accessed after the replacement. We thus introduced a *swap* operation [15] into the Stainless verifier. In addition to array element swap [12], Stainless now offers a **Cell** class that encapsulates a mutable variable and offers a **swap** operation to swap the content of two cells. This construct enlarges the expressiveness of Stainless without the need for aliasing and enables the implementation of a resizable data structure on top of a fixed-size one.

Table 1. Lines of code for program, as well as specification and proof. We use many ghost functions to express induction proofs. When a function has many arguments, we typically typeset each argument on a separate line, contributing to line counts.

Class	Program LOC	Proof+spec. LOC	Total LOC
ListLongMap	156	678	834
MutableLongMap	409	7'358	7'767

3.3 Finding and Confirming a Bug in the Original Implementation

During the verification, we discovered that the `repack` function does not satisfy the specification stated in its documentation. The documentation states that the map can accommodate up to 2^{30} values (preferably not more than 2^{29}) [25]. However, a number of keys greater than or equal to 2^{28} makes `repack` loop forever. The bug arises in the computation of the new `mask` and is an integer overflow: a mask candidate is reduced while it is $> _size * 8$ (where `_size` is the number of keys stored in the array). However, if `_size * 8` overflows, i.e., $_size \geq 2^{28}$, the mask candidate is reduced below `_size`. The new array then cannot accommodate all the keys. We fix the bug by modifying the loop condition and then prove that the function always returns a large enough and valid mask. Despite the small scope hypothesis [16] and claims in [8], we do not expect that bounded model checking would have discovered this bug, given that it occurs only after inserting so many key-value pairs.

3.4 SMT Queries

Stainless generates verification conditions (VCs) which are solved by Inox [28] using SMT solvers (here, CVC4 [3], cvc5 [2], and Z3 [9]) and incremental function unrolling. So, a sequence of calls to SMT solvers happens for each VC and solvers run in parallel in a race. Generated SMT queries [6] use algebraic datatypes and sets. They do not contain *set-logic* directive. Only the query corresponding to the winning solver is recorded for each VC, as the others might be incomplete. Stainless uses generalized arrays [21] with non-constant default values to encode generic arrays among other things. This feature is unavailable on CVC4 and not implemented in Stainless for cvc5. Hence, VCs using it can be solved only by Z3. This partially explains why Z3 is solving most queries (Fig. 2 (right)).

4 Evaluation

We run the benchmarks on an Ubuntu 20.04.6 LTS server with an Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80 GHz, 64 GB RAM.

Verification takes around 400 s when running from scratch (around 100 s when re-running with a populated verification-condition cache [10]). Figure 2 shows the VCs solving time (with cache completely disabled). Most VCs are solved quickly,

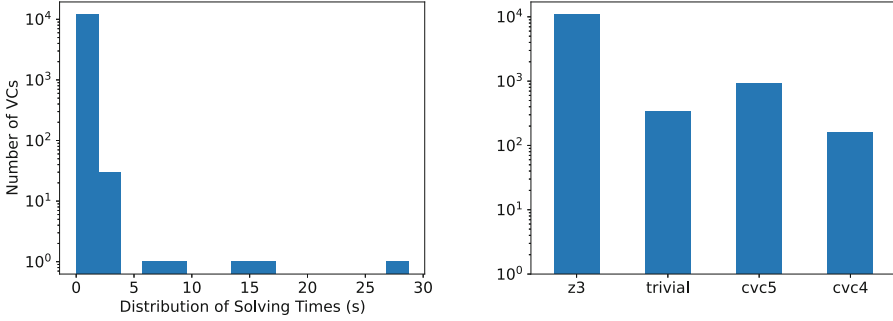


Fig. 2. Left: VC solving time distribution with Stainless cache disabled. Right: number of queries solved by each solvers. Both use a logarithmic scale.

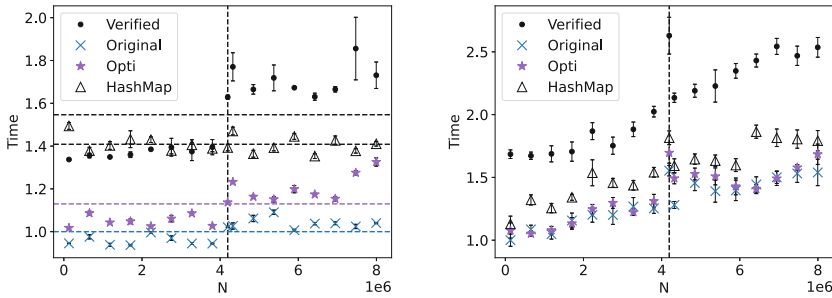


Fig. 3. Lookup of N keys in a map prepopulated with 2^{22} pairs (left) (time normalized per operation) and insertion of 2^{22} pairs (initial capacity of 16) followed by lookup of N keys (right). Horizontal lines show the average. The black vertical lines show 2^{22} . The error bars show the 95% CI. The time on the y-axis is normalized with respect to the first data point of the original map.

with a mean and a median of around 0.16 and 0.1 s, respectively. The cumulative solving time is 1’937s. Only 3 VCs need more than 10s in Stainless, with one VC capping at 29s out of 12’122 VCs. When calling the solver directly on the generated SMT-LIB files, the cumulative solving time falls to 407s. This likely shows the overhead of the unrolling in Inox [28], which is especially visible for fast VCs. Figure 2 also shows the distribution of VCs solved by each solver.

We compare the performance of our verified implementation to the original [26], the general `HashMap` of the Scala standard library [24], and an optimized version (denoted `Opti`) that changes the verified implementation to use `Array[AnyRef]` for `_values`. We use `Long` as the type of stored values. We consider three scenarios: looking up keys in a pre-populated map, populating the map first, then looking up keys, and populating the map with all pairs, removing half of the keys, and inserting all pairs again before looking up keys. Results are in Fig. 3 and Fig. 4. Our verified `LongMap` is around 1.5× slower than the original implementation for lookups only, see Fig. 3 (left). The performance gap is similar

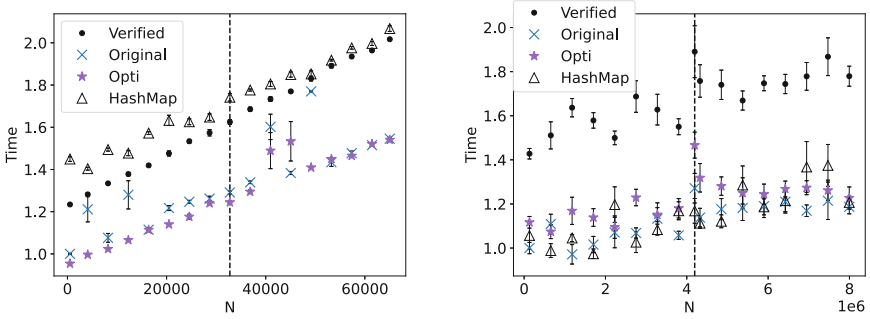


Fig. 4. Total time to lookup N keys and: **(left)** insert 2^{15} pairs with initial capacity 2^{17} , or **(right)** insert 2^{22} pairs, remove 2^{21} , and insert 2^{22} again, with initial capacity of 16. The black vertical line shows 2^{15} (left) and 2^{22} (right). The error bars show the 95% CI. The time on the y-axis is normalized with respect to the original map.

when taking the population process into account (Fig. 3 (right)). We argue that this is acceptable. Indeed, the `LongMap` is the fastest map we know in the Scala ecosystem. As shown by Fig. 4, the performance of our verified implementation is comparable to the Scala `HashMap` (better in some scenarios).

Consequences of Adapting for Verifiability. To understand the impact of pointer indirection in the `_values` array (Sect. 2.1), we modified our verified implementation to use `Array[AnyRef]` like the original (abandoning the proof). The results are shown as `Opti` in the figures, with performance close to the original one, indicating that this indirection was indeed responsible for the overhead. Similarly, in our version, creating `_values` and `_keys` arrays relies on `Array.fill`, which writes all values and is slower than constructing an array of `SPSVERBc48s` in the original implementation. Therefore, the verified `repack` operation is slower than the original, see Fig. 4 (right). As shown by Fig. 4 (left), without resizing, the performance is similar to `HashMap`, suggesting the impact of `Array.fill`. Calls to `repack` are infrequent, so this performance loss is limited. Finally, as witnessed by the `Opti` implementation performance being close to the original, there is limited performance impact of the way seek functions pass information to the caller, and of counter checks for loop termination (Sect. 2.1).

5 Conclusion

We verified `LongMap` from the Scala standard library, a mutable hash table with `Long` keys, employing open addressing and non-linear probing. This led us to identify and fix a bug in the original library implementation. The performance evaluation of our verified implementation against the original shows a slowdown of around 1.5. The changes we needed to perform for verifiability point to directions for further improving verification support for efficient Scala constructs.

References

- Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016). <https://doi.org/10.1007/978-3-319-49812-6>, <http://link.springer.com/10.1007/978-3-319-49812-6>
- Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 415–442. Springer International Publishing, Cham (2022)
- Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
- Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 60–80. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
- Bucev, M., Kunčak, V.: Formally verified Quite OK Image format. In: Formal Methods in Computer-Aided Design (FMCAD) (2022)
- Chassot, S., Kunčak, V.: Verifying a Realistic Mutable Hash Table Case Study (Short Paper) (Artifact) (Apr 2024). <https://doi.org/10.5281/zenodo.11079220>
- Commit: New mutable hash map with long keys. <https://github.com/scala/scala/commit/05aedd936e7e7bcf0fa2443abd58b39732f173a9>
- De Boer, M., De Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK’s identity hash map implementation. Formal Aspects Comput. **35**(3), 18:1–18:26 (Sep 2023). <https://doi.org/10.1145/3594729>, <https://dl.acm.org/doi/10.1145/3594729>
- de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
- Guilloud, S., Bucev, M., Milovančević, D., Kunčak, V.: Formula normalizations in verification. In: Computer-Aided Verification (CAV) (2023)
- Gutttag, J.V.: Abstract data type and the development of data structures. Commun. ACM **20**(6), 396–404 (1977). <https://doi.org/10.1145/359605.359618>
- Hamza, J., Felix, S., Kunčak, V., Nussbaumer, I., Schramka, F.: From verified Scala to STIX file system embedded code using Stainless. In: NASA Formal Methods (NFM), p. 18 (2022). <http://infoscience.epfl.ch/record/292424>
- Hamza, J., Voirol, N., Kunčak, V.: System fr: formalized foundations for the stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). <https://doi.org/10.1145/3360592>
- Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., Parno, B.: Storage Systems are Distributed Systems (So Verify Them That Way!). In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2020)
- Harms, D.E., Weide, B.W.: Copying and swapping: influences on the design of reusable software components. IEEE Trans. Software Eng. **17**(5), 424–435 (1991). <https://doi.org/10.1109/32.90445>
- Jackson, D., Damon, C.A.: Elements of style: analyzing a software design feature with a counterexample detector. ACM SIGSOFT Softw. Eng. Notes **21**(3), 239–249 (May 1996). <https://doi.org/10.1145/226295.226322>, <https://dl.acm.org/doi/10.1145/226295.226322>

17. Jahob Hashtables Codebase. <https://github.com/epfl-lara/jahob/tree/master/examples/containers/hashtable>
18. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (Feb 2007). <http://hdl.handle.net/1721.1/38533>
19. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. *ACM SIGPLAN Notices* **52**(1), 330–343 (Jan 2017). <https://doi.org/10.1145/3093333.3009874>, <https://dl.acm.org/doi/10.1145/3093333.3009874>
20. Milovančević, D., Kunčak, V.: Proving and disproving equivalence of functional programming assignments. In: *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)* (2023)
21. de Moura, L.M., Bjørner, N.S.: Generalized, efficient array decision procedures. In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pp. 45–52. *IEEE* (2009). <https://doi.org/10.1109/FMCAD.2009.5351142>
22. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): *Isabelle/HOL*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
23. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. *Formal Aspects Comput.* **30**(5), 495–523 (Sep 2018). <https://doi.org/10.1007/s00165-017-0435-1>, <https://doi.org/10.1007/s00165-017-0435-1>
24. *HashMap Scala Standard Library*. <https://scala-lang.org/api/3.3.1/scala/collection/mutable/HashMap.html>
25. *LongMap Specification*. <https://github.com/scala/scala/blob/263e5bd60d9c3947d8d17b7d8769a4b94f6865c7/src/library/scala/collection/mutable/LongMap.scala#L36>
26. *LongMap Implementation - Standard Library*. <https://github.com/scala/scala/blob/948ed0b60466803f404d5f24a3afc0a89c06ffc1/src/library/scala/collection/mutable/LongMap.scala>
27. Team, T.C.D.: *The Coq Proof Assistant* (Jun 2023). <https://doi.org/10.5281/ZENODO.1003420>. <https://zenodo.org/record/1003420>, language: en
28. Voirol, N.C.Y.: *Verified functional programming*, p. 229 (2019). <https://doi.org/10.5075/epfl-thesis-9479>
29. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: *ACM SIGPLAN Conference Programming Language Design and Implementation (PLDI)* (2008)



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

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





Booleguru, the Propositional Polyglot (Short Paper)

Maximilian Heisinger^(✉) , Simone Heisinger , and Martina Seidl 

Johannes Kepler University Linz, Linz 4040, Austria
{maximilian.heisinger,simone.heisinger,martina.seidl}@jku.at

Abstract. Recent approaches on verification and reasoning solve SAT and QBF encodings using state-of-the-art SMT solvers, as it “makes implementation much easier”. The ease-of-use of these solvers make SAT and QBF solvers less visible to users of solvers—who are maybe from different research communities—potentially not exploiting the power of state-of-the-art tools. In this work, we motivate the need to build bridges over the widening solver-gap and introduce BOOLEGURU, a tool to convert between formats for logic formulas. It makes SAT and QBF solvers more accessible by using techniques known from SMT solvers, such as advanced Python interfaces like Z3Py and easily generatable languages like SMT-LIB, integrating them to our conversion tool. We then introduce a language to manipulate and combine multiple formulas, optionally applying transformations for quickly prototyping encodings. BOOLEGURU’s advanced scripting capabilities form a programming environment specialized for Boolean logic, offering a more efficient way to develop novel problem encodings.

Keywords: SAT · QBF · SMT · DIMACS · QCIR · SMTLIB2 · AIGER

1 Introduction

Numerous recent publications with encodings of problems into SAT and QBF do not use SAT or QBF solvers directly [6, 16, 18]. SMT solvers, often the feature-rich and popular state-of-the-art solver Z3 [8], are used instead, as it “makes implementation much simpler” [17], although no theory reasoning is involved. Z3’s programming API or the Common Lisp compatible SMT-LIB standard [3] are well documented and regarded by many as easy to use. While this ease of use leads to wide adoption and fast results, adapting encodings to use less general solving backends that are potentially more efficient for the problem at hand remains hard, e.g. switching from SMT solving to using a less general SAT solver. Researchers focus on optimizing their encodings against an SMT solver’s performance characteristics, instead of testing them against many different (also

This work was supported by the LIT AI Lab funded by the State of Upper Austria.

© The Author(s) 2024

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 315–324, 2024.

https://doi.org/10.1007/978-3-031-63498-7_19

Table 1. Formula formats that are optimized (✓), usable (≈), or unusable (✗) for encoding the respective problem

Problem	DIMACS	QDIMACS	QCIR	AIGER	SMTLIB2
SAT Solving	✓	≈	≈	≈	≈
QBF Solving	✗	✓	✓	✗	≈
AIG Solving	✗	✗	≈	✓	✓
SMT Solving	✗	✗	✗	✗	✓

Table 2. File formats and their capabilities.

Format	<i>Non-CNF</i>	<i>Quantifiers</i>	<i>Non-Prenex</i>	<i>Structure-Sharing</i>
DIMACS	✗	✗	✗	✗
QDIMACS	✗	✓	✗	✗
AIGER	✓	✗	✗	✓
Limboole	✓	✓	✓	✗
QCIR	✓	✓	✓	✓
SMTLIB	✓	✓	✓	✓

non-SMT) solvers. We consider the transformation of formulas into conjunctive normal form (CNF) required for SAT and QBF solvers to be non-trivial, especially for beginners. Seemingly bad intermediary results are discarded, as the effort required to re-encode the problem to be solvable with SAT or QBF solvers is too large to be spent during prototyping, effectively forming a *solver gap*. We want to bridge over this gap and reduce the friction involved with testing other solvers outside the SMT world, without extensive modifications to encoding generators. In this work, we analyze what features are required to build this bridge and develop BOOLEGURU, a polyglot for (quantified) Boolean logic. Our tool is available under the permissive MIT license at:

<https://github.com/maximaximal/booleguru>

1.1 Bridging the Solver Gap with Propositional Logic

In order to build a bridge over the solver gap described above, we first have to identify it. When encoding problems into logic, one also has to decide which format to encode into. The encoding itself is then typically accomplished with some encoder program, which is closely tied to the problem to be encoded. Changing the output format of an encoder involves considerable effort, as encoders are typically tailored to the output formats they were designed to support. As encoders grow in complexity, communities form around them, while still relying on the original output format. The decision at the beginning of an encoder’s development then influences the newly formed community, as changing their encoder to generate a different output format involves considerable effort. We



Fig. 1. Booleguru Architecture, Transformers may be arbitrarily combined.

therefore identify gaps in the solving landscape opening between the different input formats of different solvers for different problems. Table 1 lists a selection of different problems with their associated dominant file formats. Table 2 lists features offered by each format. All of them offer ways to encode propositional logic in CNF, with some of them extending it with quantifiers over variables (\forall , \exists). More advanced formats also allow encoding formulas in Non-CNF, i.e., formulas built from expressions and more complex logical operators. If a format supports quantifiers, they may always be added as a prefix to the formula. While every formula with embedded quantifiers can be prenexed to be in such a prenex form, some formats also allow encoding formulas in non-prenex form, extending a format’s expressiveness. Some formats also allow structure-sharing, which lets problems reference sub-expressions multiple times, without repeating them. The overlapping capabilities of different formats suggest that a conversion tool has to be able to process all of them, and to serialize complex features into less complex formats, where supported.

1.2 Related Work

Other communities already went through this bridge-building effort in order to reduce duplicated work and advance their fields. One of the biggest examples are the researchers of the machine learning community, who commonly use libraries like PyTorch [15], SciPy [21], and Pandas [20]. This allows others to use new innovations in these libraries, such as newly added learning algorithms or properties in PyTorch, or better storage formats in Pandas.

Multiple conversion tools already exist for QBF [9, 13, 19]. While all of these tools convert between specific formats, no tool tries to encompass multiple conversion or combination capabilities. Some SMT solvers are able to read multiple input formats [2, 7, 8], with all supporting SMT-LIB2, the format favored by the SMT-Competition [22]. SMT solvers do not offer to combine multiple formulas seamlessly. BOOLEGURU fills this niche and provides such a convert & combine capability, while also enabling a unique development environment to create new formulas. It enables previously tedious comparisons between different solvers solving similar problems, like SMT and QBF solvers, as shown in Fig. 2.

2 Booleguru, the Propositional Multitool

After introducing the overlaps between file formats and solving communities, we now introduce our conversion tool: BOOLEGURU. As shown in the architecture diagram in Fig. 1, it consists of readers, transformers, and serializers for

propositional logic and extensions. Inputs in arbitrary formats may be read, modified, and then serialized in the same or a different file format. This section first describes how Booleguru stores propositional formulas in memory, so that all capabilities described in Table 2 can be provided. We then introduce features intended for working with formulas.

2.1 Representing Propositional Formulas in Memory

In order to be accepted as an efficient tool to work with propositional formulas, they have to be both fast to create and to traverse. While this is true for a tool in any problem domain, the lower expressiveness of propositional logic compared to bit-vectors or more complex theories leads to large formulas with many nodes, relying on tools with especially high throughput. For this, they are stored in a directed-acyclic-graph (DAG), enabling structure-sharing. Each node in the DAG is either a variable, a unary (negation), or a binary operation (and, or, etc.). A node is stored in a struct of 16B, which (on most architectures) exactly fits into a single cache-line. The remaining bits to fill the cache line are occupied by structural information of the expression and user-writable extra data to be stored within the DAG, which speeds up transformers that need to store temporary data on nodes. Beside the user writable data, nodes stay constant over the whole execution.

References between nodes are stored as 32bit unsigned integers, which are evaluated relatively to the nodes of a whole formula. When creating a formula, a hash table is used to check if a given node already exists, and if it does not, a new node is appended at the back of the node array. References to previous nodes are immutable, which enables cheaply appending new expressions that are composed of others. Expressions may only reference other expressions with IDs smaller than themselves, cycles imply a malstructured formula. Traversing this DAG does not involve hash lookups or pointer indirections, as every reference can be resolved directly through the child's index in the array. Information about a sub-expression is collected during insertion of a new node, removing the requirement to scan the DAG in order to check for commonly required structural information. The 32bit references make traversal very efficient, but limits formula sizes to $2^{32} - 1$ nodes. We will provide a compile-time switch to increase reference sizes in a future version.

2.2 Parsing Formulas

We already implemented several parsers:

- (Q)DIMACS
- QCIR [13]
- AAG (AIGER [5])
- SMT-LIB [3]
- Z3Py
- CLI
- generic infix logic using `&`, `|`, `<->`, etc. (Limboole)

The readers are mostly implemented using the ANTLR parser generator [14]. While slower than hand-written parsers, the library allowed us to iterate faster

during development and add more input languages with a shared base. Some parsers are hand-written, and performance critical ones are incrementally optimized to a specialized implementation. Each parser produces a reference to an expression inside a shared expression manager. Multiple expressions can then be combined into new ones, disregarding their source format. The command-line parsing is also fully done using an ANTLR parser producing an expression, in order to provide a language for composing formulas from multiple files or scripts.

2.3 Transforming Formulas

Transformers are the umbrella term for functions that work on one or more expressions passed to them. They may return new expressions built from their inputs and may be chained together. They are either implemented in native C++, Lua [12], or Fennel¹, with Lua and Fennel possibly supplied at runtime by a user without re-compiling BOOLEGURU. Several transformations are already implemented, with more being added in future releases. The list below uses the *Colon Operator* (`:op`) notation, which is transformed into Fennel function calls during CLI parsing. Each such transformation can be supplied to the BOOLEGURU CLI, where they strongly bind (stronger than binary operators) to the expression preceding them. Generating transformers without expressions as inputs have to be written without an expression preceding them, akin to variables.

<code>:eliminate-implication</code>	<code>:tseitin</code>
Converts $a \Rightarrow b$ to $\neg a \vee b$.	Tseitin-encode a (sub-) expression into CNF, without the exponential blowup involved with
<code>:eliminate-equivalence</code>	<code>:distribute-to-cnf</code> .
Converts $a \Leftrightarrow b$ to $(a \vee \neg b) \wedge (b \vee \neg a)$	
<code>:eliminate-xor</code>	<code>:rename</code>
Converts $a \oplus b$ to $\neg a \wedge b \vee a \wedge \neg b$	Rename one or more variables in a (sub-) expression. Can take multiple arguments.
<code>:distribute-ors</code>	<code>:solve</code>
Distributes \vee into the formula and remove them from the outermost context.	Solve a (sub-) expression in CNF. Returns a conjunction of variables.
<code>:distribute-nots</code>	<code>:quanttree</code>
Distributes \neg into the formula and remove them from the outermost context or from applying to sub-expressions.	Draw a formula's quantifiers.
<code>:distribute-to-cnf</code>	<code>:unquantified</code>
Distributes operations in the formula until it is in Conjunctive Normal Form (CNF). This often entails exponential size increase.	Print all variables that are not quantified by some quantifier.
	<code>:prefixtract</code>
	Extract statistics from a formula's quantifier prefix (if it is in prenex form).

¹ <https://fennel-lang.org>.

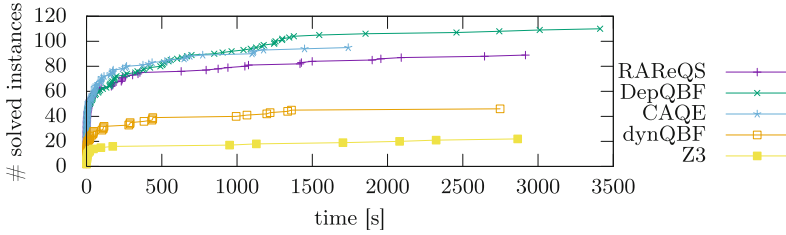


Fig. 2. Z3 and selected QBF solvers solving the QCIR track of QBFGallery 2023

:quantlist

Print the quantifier prefix (if available), merging multiple quantifiers of same type into sorted blocks.

:counterfactuals

Generate a counterfactual (parameterized).

:eqkbfk

Generate a KBKF combined with an equality formula (parameterized).

:dotter

Outputs the in-memory DAG of the formula as a `.dot` file that can be processed using *GraphViz*.

:assignment-tree

Expands the quantifier prefix into a tree over all possible variable assignments and solves each leaf assignment using SAT solver. Outputs a `.dot` file.

2.4 Serializing Formulas

After reading, combining, and transforming input formulas, they can be printed in different output formats. For this, several serializers have been developed, which are listed below. (Q)DIMACS relies on the provided Tseitin encoding by default, but one may use other methods that arrive on an expression which is tagged to be in CNF. This made BOOLEGURU a helpful tool in the QBFGallery 2023².

- (Q)DIMACS
- QCIR
- SMTLIB
- Limboole

3 Booleguru, the Programming Environment

During development of new problem encodings or crafted formulas, there is usually a step where an encoding tool or a formula generator is written [1, 4, 11]. In our experience, these tools often rely on similar primitives:

- Create a new variable.
- Compose an expression based on sub-expressions.
- Read from an external source or draw random data.

² <https://qbf23.pages.sai.jku.at/gallery/>.

- Write the formula in the desired output format.

The Lua, Fennel, and Python APIs offered by Booleguru abstract over multiple output formats and the concept of writing formulas into files. When writing a generator for a new formula, the Booleguru primitives offer the user to work directly with the formula’s AST, instead of having to generate syntax describing the AST in the target language. This makes generators more suited to change, as they are always composed of (nested) functions, each generating sub-expressions.

In addition to using BOOLEGURU as a first-class execution environment for formula generators, it may be used to reduce SMT encodings developed using Z3Py to a SAT or QBF solving problem. BOOLEGURU optionally generates a Python module that emulates the popular interface of Z3.

Lua and Fennel Interface. The Lua and Fennel interfaces are both accessible through an embedded interpreter. Lua and Fennel scripts that are already distributed as a part of BOOLEGURU are compiled ahead of time by LuaJIT³. This makes both initialization and execution of scripts as efficient as possible in the Release build of BOOLEGURU. User-supplied scripts are compiled at runtime using LuaJIT.

Python Interface. Additionally to the specialized Lua and Fennel interfaces, BOOLEGURU provides the `pybooleguru` interface which is directly modelled after the widely used *Z3Py* Python library. We observed that when using Z3Py during development of a new encoding, the jump from the SMT solver Z3 to more fundamental SAT or QBF solving becomes harder. This interface is intended to be a drop-in replacement to Z3Py, enabling the conversion of a complex Z3-specific encoder written in Python into an encoder capable of producing additional output of a different format. Python scripts may be read as inputs or a script may import `pybooleguru` instead of `z3py`.

C++ Interface. Additionally to the scripting interfaces, the C++ interface itself may also be used. Functions are provided to easily build expressions using C++, which can be useful when developing new tools in systems languages.

3.1 Command-Line Interactive Interface

The command-line interface of BOOLEGURU is also considered a programming environment, as it seamlessly merges a grammar for propositional logic in infix notation with the Scheme-based Fennel, a programming language written in prefix-notation. Each Fennel expression has to return a logical expression, which it builds using the provided primitives. By combining transformations and working with expressions, new functionality can be implemented using the CLI alone. For example, for the formula $(a \wedge b) \vee (a \wedge \neg b)$, both solutions can be extracted using the CLI:

³ <https://luajit.org/luajit.html>.

```
$ booleguru test.boole :solve
a & !b
$ booleguru test.boole ":(b-and ** (b-not (solve **)))" :solve
a & b
```

It can also be used to combine formulas while renaming e.g. a to aa :

```
$ booleguru "test.boole :rename@a@aa <-> test.boole"
(#aa ?b (aa & b)) <-> (#a ?b (a & b))
```

The CLI can also invoke parameterized custom binary operators. The # comment below is the file `my-bin.fnl` in the current working directory.

```
# (lambda my-bin [m] ((. _G m) (b-and *l* *r*) (b-or *l* *r*)))
$ ./booleguru a ::my-bin@equi b
a & b <-> a | b
```

3.2 Developing Booleguru

Our tool is implemented in C++, with some helpers provided to ease development. The build system is realized using CMake, which makes BOOLEGURU easily embeddable into other projects. All modules have test suites to be run during development. They test basic features like the expression tree, but also transformers and serializers. Defining new tests is easy and running all tests is quick and parallelizable. The build system offers a fast Release build, a slower Debug build without optimization options, and a Sanitized build with enabled address- and undefined behavior sanitizers. If available, LuaJIT is used for executing Lua and Fennel scripts, otherwise the regular Lua distribution is provided as a fallback. Booleguru also natively supports to be built as a WebAssembly executable, making it runnable in browsers, including Lua and Fennel scripts.

Embedded Fuzzer. Transformers that process operations may also be fuzzed using the LLVM *libFuzzer* integration. BOOLEGURU has to be built in the fuzzing mode, which creates the specialized `booleguru-fuzzer` binary. The command-line has to be provided via the environment variable `BOOLEGURU_ARGS`, with a `fuzz` file that serves as the injection point for fuzzed inputs. Arbitrary transformations may then be performed on the expressions, which are called with every iteration of the fuzzer. BOOLEGURU's embedded mutator (inspired by the mutator of Google Protocol Buffers) randomly creates arbitrary many input structures until unexpected stops are encountered. The fuzzing capability was used extensively during development of the transformers. The resulting inputs can be displayed in Limboole format using the `booleguru-print-corpus` tool.

4 Conclusion

We developed the propositional polyglot BOOLEGURU, which can be used to convert between several widely used logic formats, to transform or combine formulas,

and to develop new encodings more efficiently. We discussed the requirements our tool has to fulfill and introduced the implementation based on these requirements. Finally, we explained how BOOLEGURU is used to generate new encodings, using the embedded Lua, Fennel, and Python scripting support. BOOLEGURU already proved itself as a valuable tool during the QBFGallery 2023, for revisiting quantifier shifting in QBF [10], and other projects.

References

1. Amendola, G., Ricca, F., Truszczyński, M.: A generator of hard 2QBF formulas and ASP programs. In: Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (2018)
2. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barret, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6, May 2021. www.SMT-LIB.org
4. Beyersdorff, O., Pulina, L., Seidl, M., Shukla, A.: QBFFam: a tool for generating QBF families from proof complexity. In: Li, C.-M., Manyà, F. (eds.) SAT 2021. LNCS, vol. 12831, pp. 21–29. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_3
5. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
6. Bonnah, E., Nguyen, L., Hoque, K.A.: Motion planning using hyperproperties for time window temporal logic. *IEEE Robot. Autom. Lett.* **8**, 1–8 (2023)
7. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_16
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Hecking-Harbusch, J., Tentrup, L.: Solving QBF by abstraction. In: *Electronic Proceedings in Theoretical Computer Science* (2018). <https://doi.org/10.4204/eptcs.277.7>
10. Heisinger, S., Heisinger, M., Rebola-Pardo, A., Seidl, M.: Quantifier shifting for quantified boolean formulas revisited. In: Benz Müller, C., Heule, M., Schmidt, R. (eds.) *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings*. LNCS, vol. 14739, pp. 325–343. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-63498-7_20
11. Heisinger, S., Seidl, M.: True crafted formula families for benchmarking quantified satisfiability solvers. In: Dubois, C., Kerber, M. (eds.) *CICM 2023*. LNCS, vol. 14101p, pp. 291–296. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-42753-4_20
12. Ierusalimsky, R.: *Programming in Lua*. Roberto Ierusalimsky (2006)
13. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: *AAAI Workshop: Beyond NP*. AAAI Technical report, vol. WS-16-05. AAAI Press (2016)

14. Parr, T.: The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference*, pp. 1–326 (2013)
15. Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035. Curran Associates, Inc. (2019)
16. Peham, T., Brandl, N., Kueng, R., Wille, R., Burgholzer, L.: Depth-optimal synthesis of Clifford circuits with SAT solvers (2023)
17. Saaltink, C., Nicoletti, S.M., Volk, M., Hahn, E.M., Stoelinga, M.: Solving queries for Boolean fault tree logic via quantified sat. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, pp. 48–59. FTSCS 2023, Association for Computing Machinery (2023). <https://doi.org/10.1145/3623503.3623535>
18. Schwarzová, T., Strejcek, J., Major, J.: Reducing acceptance marks in Emerson-lei automata by QBF solving. In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, Alghero, Italy. LIPIcs*, vol. 271, pp. 23:1–23:20 (2023). <https://doi.org/10.4230/LIPICS.SAT.2023.23>
19. Seidl, M., Lonsing, F., Biere, A.: qbf2epr: a tool for generating EPR formulas from QBF. In: *Third Workshop on Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30–July 1, 2012. EPIc Series in Computing*, vol. 21, pp. 139–148. EasyChair (2012). <https://doi.org/10.29007/2B5D>
20. The pandas development team: pandas-dev/pandas: Pandas, February 2020. <https://doi.org/10.5281/zenodo.3509134>
21. Virtanen, P., et al.: SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, pp. 261–272 (2020). <https://doi.org/10.1038/s41592-019-0686-2>
22. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015–2018. *J. Satisf. Boolean Model. Comput.* **11**(1), 221–259 (2019). <https://doi.org/10.3233/SAT190123>





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

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





Quantifier Shifting for Quantified Boolean Formulas Revisited

Simone Heisinger¹(✉) , Maximilian Heisinger¹ , Adrian Rebola-Pardo^{1,2} ,
and Martina Seidl¹ 

¹ Institute for Symbolic Artificial Intelligence, JKU Linz, Linz, Austria
{simone.heisinger,maximilian.heisinger,adrian.rebola_pardo,
martina.seidl}@jku.at

² Institute for Logic and Computation, TU Vienna, Vienna, Austria

Abstract. Modern solvers for quantified Boolean formulas (QBFs) process formulas in prenex form, which divides each QBF into two parts: the quantifier prefix and the propositional matrix. While this representation does not cover the full language of QBF, every non-prenex formula can be transformed to an equivalent formula in prenex form. This transformation offers several degrees of freedom and blurs structural information that might be useful for the solvers. In a case study conducted 20 years back, it has been shown that the applied transformation strategy heavily impacts solving time. We revisit this work and investigate how sensitive recent QBF solvers perform w.r.t. various prenexing strategies.

Keywords: Quantified Boolean Formulas · Prenexing · Normal Form Transformation

1 Introduction

Quantified Boolean formulas (QBFs), the extension of propositional formulas with quantifiers over the Boolean variables, have many applications in formal verification, synthesis, and artificial intelligence [28]. Over the last 25 years, many efficient QBF solvers have been developed [2], with clear tendency towards QBFs in prenex conjunctive normal form (PCNF). A QBF in PCNF has the form $Q_1x_1 \dots Q_nx_n \cdot \phi$ where $Q_i \in \{\forall, \exists\}$ and ϕ is a propositional formula in conjunctive normal form. In general, encodings do not result in formulas of this structure, because of recursive definitions in the encoding or from optimizations that try to minimize the scope of variables. Origins for a non-CNF structure can be for example the use of equivalences or xors in the encoding. Therefore two transformations are required: (1) *prenexing* which shifts the quantifiers outside of the formula, and (2) transformation of the quantifier-free formula to CNF. The latter is efficiently achieved by applying the QBF-variant of the well known Tseitin transformation [30] or the optimized Plaisted-Greenbaum transformation [24]. In this work, we focus on the prenexing.

*This work was supported by the LIT AI Lab funded by the state of Upper Austria and by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG11005].

© The Author(s) 2024

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 325–343, 2024.

https://doi.org/10.1007/978-3-031-63498-7_20

Without loss of generality, formulas can be assumed to be in negation normal form (i.e. negation symbols only occur in front of variables) and cleansed (i.e. no variable occurs both bound and free, and every variable is quantified at most once). Under these conditions, prenexing is achieved by the following two rules:

$$(\mathbf{Q}x.\varphi) \circ \varphi' \Leftrightarrow \mathbf{Q}x.(\varphi \circ \varphi') \quad \varphi \circ (\mathbf{Q}x.\varphi') \Leftrightarrow \mathbf{Q}x.(\varphi \circ \varphi')$$

with $\mathbf{Q} \in \{\forall, \exists\}$ and $\circ \in \{\wedge, \vee\}$. The formula structure imposes an ordering of the quantifiers based on the subformula relation. Quantifiers from independent parts of a formula can be freely ordered. For example, the formula $\forall x.(\exists y.(y \vee x)) \wedge (\forall z.(z \vee \neg x))$ has prenex forms $\forall x.\exists y.\forall z.\phi$ or $\forall x.\forall z.\exists y.\phi$ where $\phi = (y \vee x) \wedge (z \vee \neg x)$. Hence, a prenex form is not uniquely determined.

Egly et al. [6] suggested four different prenexing strategies that minimize the number of quantifier alternations in the prefix. Empirically, they showed that the selected prenexing strategy impacts solving performance. In this work, we revisit those prenexing strategies and give a concise formalization, which the original work lacked. We show that the original four strategies disambiguate into six unique prenexing strategies when enforcing a minimal number of quantifier alternations, and we present a tool that implements those strategies. To evaluate the impact of prenexing on modern solvers we reimplemented the generator for encoding nested counterfactuals and performed extensive experiments with these formulas and formulas from the QBF Eval'08 in which a non-prenex track was organized.

2 Preliminaries

The set of *quantified Boolean formulas* $QF(X)$ over variables X is defined as follows: (1) $\top, \perp, x, \neg x, \neg\top, \neg\perp \in QF(X)$ if $x \in X$, (2) $\varphi \vee \varphi', \varphi \wedge \varphi' \in QF(X)$ if $\varphi, \varphi' \in QF(X)$, (3) if $\varphi \in QF(X)$, then $\forall x.\varphi, \exists x.\varphi \in QF(X)$ with $x \in X$.¹ In QBF $\mathbf{Q}x.\varphi$ with $\mathbf{Q} \in \{\forall, \exists\}$, the subformula φ is called the scope of variable $x \in X$ and x is said to be bound by quantifier \mathbf{Q} . If variable x occurs in QBF φ , but φ does neither contain $\exists x$ nor $\forall x$ then x is free in φ . The set of all free variables of a QBF φ is denoted by $free(\varphi)$. A QBF without free variables is called closed. In the following, we assume that each variable is in the scope of at most one quantifier and that each variable occurs either free or bound, but not both in a formula. We call such formulas *cleansed*. The semantics of a QBF φ is defined by the interpretation function $[\cdot]_\sigma: QF(X) \rightarrow \mathbb{B}$ where $\mathbb{B} = \{\mathbf{1}, \mathbf{0}\}$ and $\sigma: free(\varphi) \rightarrow \mathbb{B}$ is an assignment to the free variables of φ . Then $[\top]_\sigma = \mathbf{1}$, $[\perp]_\sigma = \mathbf{0}$, for any $x \in X$, $[x]_\sigma = \sigma(x)$ and $[\neg v]_\sigma = 1 - [v]_\sigma$ for $v \in X \cup \{\top, \perp\}$. Furthermore, $[\varphi_1 \wedge \varphi_2]_\sigma = \min\{[\varphi_1]_\sigma, [\varphi_2]_\sigma\}$, and $[\varphi_1 \vee \varphi_2]_\sigma = \max\{[\varphi_1]_\sigma, [\varphi_2]_\sigma\}$. Finally, $[\exists x.\varphi]_\sigma = \max\{\varphi[x|\top], \varphi[x|\perp]\}$ and $[\forall x.\varphi]_\sigma = \min\{\varphi[x|\top], \varphi[x|\perp]\}$ where $\varphi[x|t]$ denotes the QBF obtained by substituting a variable x by a truth constant $t \in \{\top, \perp\}$ in φ . Two QBFs $\varphi, \varphi' \in QF(X)$ are equivalent if $[\varphi]_\sigma = [\varphi']_\sigma$ for any assignment σ .

¹ For simplicity, we assume negations only in front of variables and truth constants.

Definition 1. The propositional skeleton φ_{psk} of QBF $\varphi \in QF(X)$ is defined as follows:

$$\varphi_{\text{psk}} = \begin{cases} x & \text{if } \varphi = x, & x \in X \cup \{\top, \perp\} \\ \neg x & \text{if } \varphi = \neg x, & x \in X \cup \{\top, \perp\} \\ \varphi'_{\text{psk}} \circ \varphi''_{\text{psk}} & \text{if } \varphi = \varphi' \circ \varphi'', \circ \in \{\wedge, \vee\} \\ \varphi'_{\text{psk}} & \text{if } \varphi = \text{QV}.\varphi', \text{ Q} \in \{\forall, \exists\} \end{cases}$$

We say that a QBF φ is of the form $\text{QV}.\varphi'$ for a set of variables V whenever $\varphi = \text{Q}x_1 \dots \text{Q}x_n.\varphi'$ for some enumeration x_1, \dots, x_n of V . A QBF $\varphi \in QF(X)$ is in prenex form if it has the structure $\Pi.\phi$ where $\Pi = \text{Q}_1x_1 \dots \text{Q}_nx_n$ is a quantifier prefix with $Q_i \in \{\forall, \exists\}$, $x_i \in X$ and $x_i \neq x_j$ for $i \neq j$ and ϕ is a propositional formula. If ϕ is also in conjunctive normal form (CNF), then φ is in *prenex conjunctive normal form* (PCNF). A propositional formula is in CNF if it is a conjunction of clauses. A clause is a disjunction of literals and a literal is a variable or a negated variable. Obviously, $(\Pi.\phi)_{\text{psk}} = \phi$ for a PCNF formula $\Pi.\phi$.

Proposition 1. Consider QBFs φ, φ' , a quantifier $\text{Q} \in \{\forall, \exists\}$, a connective \circ , and variables x, y .

1. If $\varphi \circ (\text{Q}x.\varphi')$ is cleansed, then $\text{Q}x.(\varphi \circ \varphi')$ is cleansed, and both formulas are equivalent.
2. If $\text{Q}x.\text{Q}y.\varphi$ is cleansed, then $\text{Q}y.\text{Q}x.\varphi$ is cleansed, and both formulas are equivalent.

Forests, trees and partial orders. We will oftentimes regard trees and forests as partially ordered sets. In particular, we define a *forest* as set T equipped with a partial ordering \leq such that for all elements $x \in T$, the subset $\{y \in T \mid y \leq x\}$ is totally ordered. When T is finite, this definition appropriately models the recursive concept of a forest: the elements of T are nodes, and $x \leq y$ if y is a descendant of x . We say that x is *covered* by y whenever $x \leq y$ and there is no $z \in T$ with $x < y < z$. When regarding T as a forest, this means that x is the parent of y . A forest T is a tree if, additionally, for any two elements $x, y \in T$ there is another element $z \in T$ with $z \leq x$ and $z \leq y$. For a finite T , this implies that T has a least element, which corresponds to its root.

Given a forest T , we call a list x_1, \dots, x_n a *path* in T if for all $1 \leq i < j \leq n$ we have $x_i, x_j \in T$ and $x_i < x_j$. The *height* of a forest T is defined as

$$\text{ht}(T) = \max\{n \geq 0 \mid \text{there is a path } x_1, \dots, x_n \text{ in } T\}.$$

For a node $x \in T$, we define its *lower bounds* as $T^x = \{y \in T \mid y \leq x\}$ and its *upper bounds* as $T_x = \{y \in T \mid y \geq x\}$.

Parity-based functions. We will use a parity-based version of the floor and ceiling functions. Intuitively, $\lfloor n \rfloor_k$ (resp. $\lceil n \rceil_k$) rounds n down (resp. up) to the closest integer with the same parity as k . Formally, for integers $n, k \in \mathbb{Z}$ we define:

$$\begin{aligned} \lfloor n \rfloor_k &= \max\{m \in \mathbb{Z} \mid m \leq n \text{ and } m - k \text{ is even}\} \\ \lceil n \rceil_k &= \min\{z \in \mathbb{Z} \mid z \geq n \text{ and } z - k \text{ is even}\} \end{aligned}$$

Direction-parametric operators. At several points our ordering-based definitions will depend on a *direction* parameter $\dagger \in \{\uparrow, \downarrow\}$. Intuitively, \uparrow -labeled operators use a reverse ordering, while \downarrow -labeled operators use an unmodified ordering. In particular, we define the following operators:

$$\begin{array}{llllll}
 \leq^{\uparrow} & \text{is} & \geq & \leq^{\downarrow} & \text{is} & \leq & \text{(and similarly for } \geq^{\dagger}, <^{\dagger}, >^{\dagger}\text{)} \\
 \min^{\uparrow} & \text{is} & \max & \min^{\downarrow} & \text{is} & \min & \text{(and similarly for } \max^{\dagger}\text{)} \\
 [\dots]^{\uparrow}_k & \text{is} & [\dots]_k & [\dots]^{\downarrow}_k & \text{is} & [\dots]_k & \text{(and similarly for } [\dots]^{\dagger}_k\text{)} \\
 T_x^{\uparrow} & \text{is} & T^x & T_x^{\downarrow} & \text{is} & T_x & \text{(and similarly for } T_x^{\dagger}\text{)}
 \end{array}$$

3 Related Work

Already 20 years back it has empirically been shown that quantifier shifting has a severe impact on the solving performance [6] of QBF solvers. In this work, four different prenexing strategies were introduced that intuitively result in the smallest number of possible quantifier alternations. The authors noted that the presented strategies “leave room for different [prenexing] variants”. In this work, we close this gap by providing a concise formalization of quantifier shifting.

The observation that the prenexing strategy impacts solving performance motivated development of several non-prenex non-CNF solvers [7, 8, 15, 29]. With the rise of efficient preprocessing for PCNF formulas and a focus on applications with few quantifier alternations, however, solver development focused on formulas in prenex form. To deal with the information loss induced by quantifier shifting, solvers were introduced that employ dependency schemes [27] to (re-)discover and exploit variable independencies [16, 22], i.e., those solvers recover information on quantifier dependencies that is hidden in the prefix. Reeves et al. presented an approach to move Tseitin variables from the inner-most quantifier block to the outer-most possible position in the quantifier prefix [26]. The exact position is determined by the variables occurring in the formula defined by the Tseitin variable. With this reordering, they observe a considerable speed-up in solver performance. Lonsing and Egly evaluated the impact of the number of quantifier alternations on recent QBF solvers [18]. In their experiments, they established a correlation between different solving paradigms like expansion or QCDCL (see [2] for a detailed discussion of such proof systems) and the number of quantifier alternations. Also, proof-theoretical investigations [1] identify the number of quantifier alternations as source of hardness for practical solving. However, to the best of our knowledge, there is no recent study that investigates the impact of quantifier shifting on the solving behavior of state-of-the-art solvers for formulas in prenex normal form.

Nowadays there is also much interest in dependency quantified Boolean formulas (DQBF) which allow for an explicit specification of quantifier dependencies. The decision problem of these formulas is NEXPTIME-complete [23], in contrast to the PSPACE-completeness of QBFs.

4 Quantifier Shifting

In this work we aim to transform arbitrary QBF formulas φ (which are not in general prenex or in CNF) into equivalent prenex QBF formulas of the form $Q_1x_1 \dots Q_nx_n.\varphi_{\text{psk}}$. The formula φ_{psk} is not necessarily in CNF, although this can be easily achieved through the well-known Tseitin procedure.

The method we propose can be roughly summarized as follows. First, a *quantifier tree* reflecting the scope hierarchy of quantifiers in φ is constructed. Each node in this quantifier tree will then be assigned a rank with some restrictions to guarantee soundness; we call this assignment a *linearization*. Finally, the formula $Q_1x_1 \dots Q_nx_n.\varphi_{\text{psk}}$ is constructed by enumerating the bound variables x_1, \dots, x_n by rank; thanks to the restrictions on linearizations, this formula will be equivalent to φ .

Example 1. Throughout our work we will use the following QBF η as a running example:

$$\begin{aligned} \exists x_1.x_1 \wedge \left(\left(\forall y_1. \left(\exists z_1. (\neg y_1 \vee z_1) \wedge \right. \right. \right. \\ \left. \left. \left. \forall u_1. \exists v_1. (y_1 \vee \neg u_1 \vee v_1) \wedge (\neg y_1 \vee u_1 \vee \neg v_1) \right) \wedge \right. \right. \\ \left. \left. \left. (\forall z_2. (\exists u_2. \neg z_2 \vee u_2) \wedge (\forall u_3. x_1 \vee z_2 \vee u_3)) \right) \right) \vee \right. \\ \left. \left. \left. \left(\exists y_4. (y_4 \wedge \forall z_4. \exists u_4. z_4 \wedge u_4) \vee (\neg y_4 \wedge \exists z_5. \forall u_5. z_5 \wedge u_5) \right) \right) \right) \right) \end{aligned}$$

Its propositional skeleton η_{psk} is then given by:

$$\begin{aligned} x_1 \wedge \left(\left((\neg y_1 \vee z_1) \wedge (y_1 \vee \neg u_1 \vee v_1) \wedge (y_1 \vee u_1 \vee \neg v_1) \wedge \right. \right. \\ \left. \left. (\neg z_2 \vee u_2) \wedge (x_1 \vee z_2 \vee u_3) \right) \vee \right. \\ \left. (y_4 \wedge z_4 \wedge u_4) \vee (\neg y_4 \wedge z_5 \wedge u_5) \right) \end{aligned}$$

Consider the following two quantifier shifts for η :

$$\begin{aligned} \eta' &= \exists x_1. \exists y_4. \exists z_5. \forall y_1. \forall z_2. \forall u_3. \forall z_4. \forall u_5. \exists z_1. \exists u_2. \exists u_4. \forall u_1. \exists v_1. \eta_{\text{psk}} \\ \eta'' &= \exists x_1. \exists y_4. \exists z_5. \exists v_1. \forall y_1. \forall z_2. \forall u_3. \forall z_4. \forall u_5. \exists z_1. \exists u_2. \exists u_4. \forall u_1. \eta_{\text{psk}} \end{aligned}$$

While η' is equivalent to η , the QBF formula η'' is not. The intuitive reason is that in η'' the quantifier $\exists v_1$ has been pushed across quantifier alternation boundaries. This is exactly the situation our formalization will prevent.

Our formalization associates to each QBF a forest obtained by removing from its syntax tree all non-quantifier nodes. The remaining nodes are thus uniquely determined by a bound variable and a quantifier, and this forest contains all the information needed for quantifier shifting. Hence, we first define the abstract

concept of quantifier forests, and then we will show how to construct a quantifier forest from a QBF as above.

A *quantifier forest* is a triple (T, \leq, q) where (T, \leq) is a finite forest regarded as a partially ordered set (see Sect. 2) and $q : T \rightarrow \{\forall, \exists\}$. We call it a *quantifier tree* or *quantree* whenever (T, \leq) is a tree. If (T, \leq, q) is a nonempty quantree, we also define $q^*(x) = 1$ if $q(x) = q(\min(T))$ and $q^*(x) = 0$ otherwise. Given a QBF formula φ , its *associated quantifier forest* is a triple $(T_\varphi, \leq_\varphi, q_\varphi)$, where T_φ is the set of bound variables in φ , and \leq_φ and q_φ are defined recursively:

- If φ is either \top , \perp or $x \in X$, then $\leq_\varphi = \emptyset$ and $q_\varphi = \emptyset$.
- If $\varphi = \neg\varphi_0$ where $\varphi_0 \in X \cup \{\top, \perp\}$, then $\leq_\varphi = \emptyset$ and $q_\varphi = \emptyset$.
- If $\varphi = \varphi_1 \circ \varphi_2$ with $\circ \in \{\wedge, \vee\}$, then $\leq_\varphi = \leq_{\varphi_1} \cup \leq_{\varphi_2}$ and $q_\varphi = q_{\varphi_1} \cup q_{\varphi_2}$.
- If $\varphi = Qx.\varphi_0$ for a quantifier Q , then $\leq_\varphi = \leq_{\varphi_0} \cup \{(x, y) \mid y \in T_{\varphi_0}\}$ and $q_\varphi = q_{\varphi_0} \cup \{(x, Q)\}$.

Proposition 2. *Let $(T_\varphi, \leq_\varphi, q_\varphi)$ be a quantifier forest of QBF φ . If $\varphi = Qx.\varphi_0$ for a quantifier Q , then $(T_\varphi, \leq_\varphi, q_\varphi)$ is a quantree.*

Example 2. Figure 1 shows the quantree associated to the QBF η from Example 1. In general, we can only guarantee that the quantifier forest associated to a QBF is a tree when the QBF is of the form $Qx.\varphi$. For example, the quantifier forest associated with $(\forall x.x) \wedge (\exists y.y)$ is a forest with two incomparable elements x and y .

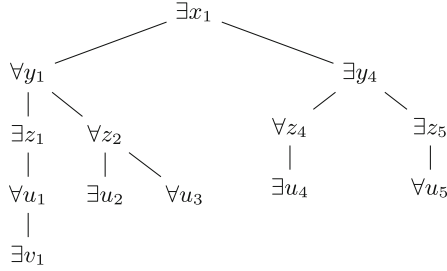
4.1 Linearizations over Quantrees

We now formalize the main object of this paper, namely the different ways the quantifiers in a formula can be rearranged into a quantifier prefix to an equivalent prenex formula. Given a QBF of the form $Qx.\varphi$ for a quantifier Q , we consider its associated quantree T . We aim to construct an equivalent prenex QBF $Q^1V_1 \dots Q^NV_N.\varphi_{\text{psk}}$ where $Q^i \neq Q^{i+1}$ for $1 \leq i < N$. To do so, each node in T (i.e. each bound variable in $Qx.\varphi$) must be mapped to a single quantifier block Q^iV_i . We call this i its *rank*. However, as shown in Example 1, assigning arbitrary ranks is unsound (i.e. the obtained prenex QBF is not equivalent to $Qx.\varphi$). We show how bound variables can be ranked while preserving soundness.

Let us consider an arbitrary quantree T . A map $f : T \rightarrow \{1, \dots, N\}$ for some $N \geq 0$ is called a *linearization* if:

1. $f(x) \leq f(y)$ for all quantree nodes $x, y \in T$ with $x \leq y$.
2. For all quantree nodes $x \in T$, $f(x)$ is odd if and only if $q^*(x) = 1$.

Consider now a QBF of the form $Qy.\varphi$ where Q is a quantifier and y is a variable, and its associated quantree (T, \leq, q) . In this case, since T is the set of bound variables in $Qy.\varphi$, a linearization $f : T \rightarrow \{1, \dots, N\}$ maps each bound variable $x \in T$ to an integer $f(x)$ we call its *rank*. A QBF ψ is called a *prenexation* of $Qy.\varphi$ via f if ψ is of the form $Q^1V_1 \dots Q^NV_N.\varphi_{\text{psk}}$ where $V_i = \{x \in T \mid f(x) = i\}$ and $Q^i = Q$ (resp. \bar{Q}) if i is odd (resp. even) for $1 \leq i \leq N$.



	\exists	\forall	\exists	\forall	\exists	
f	$f(\dots) = 1$	$f(\dots) = 2$	$f(\dots) = 3$	$f(\dots) = 4$	$f(\dots) = 5$	strategy
f_1	$x_1 y_4 z_5$	$y_1 z_2 u_3 z_4 u_5$	$z_1 u_2 u_4$	u_1	v_1	$\forall\uparrow\uparrow/\exists\uparrow\uparrow$
f_2	$x_1 y_4 z_5$	$y_1 z_2 u_3 z_4 u_5$	z_1	u_1	$v_1 u_2 u_4$	$\forall\uparrow\downarrow$
f_3	x_1	$y_1 z_2 u_3$	$z_1 y_4 z_5$	$u_1 z_4 u_5$	$v_1 u_2 u_4$	$\exists\downarrow\uparrow$
f_4	$x_1 y_4 z_5$	y_1	z_1	$u_1 z_2 u_3 z_4 u_5$	$v_1 u_2 u_4$	$\forall\downarrow\uparrow$
f_5	$x_1 y_4 z_5$	$y_1 z_2 z_4$	$z_1 u_2 u_4$	$u_1 u_3 u_5$	v_1	$\exists\uparrow\downarrow$
f_6	x_1	y_1	$z_1 y_4 z_5$	$u_1 z_2 u_3 z_4 u_5$	$v_1 u_2 u_4$	$\forall\downarrow\downarrow/\exists\downarrow\downarrow$

Fig. 1. Above, the quantree associated to the formula η from Example 1. Below, optimal linearizations for this quantree for each strategy. In each column, the variables mapped to each rank are shown; the quantifier of each block appears in the header. Note that the optimal linearizations for strategies $Q\uparrow\uparrow$ and $Q\uparrow\downarrow$ assign the same rank to Q -quantified variables; this is a consequence of Lemma 2.

Theorem 1. *Let T be the quantree associated to a QBF of the form $Qy.\varphi$. Consider a prenexation ψ of $Qy.\varphi$ via some linearization $f : T \rightarrow \{1, \dots, N\}$. Then $Qy.\varphi$ is equivalent to ψ .*

To guarantee this form $Qy.\varphi$ for an arbitrary QBF φ , we can simply introduce a fresh variable y that does not occur in φ . Obviously, φ is equivalent to $Qy.\varphi$.

Example 3. Figure 1 shows six linearizations for the quantree associated to the QBF η from Example 1 and Example 2. In that example, the quantifier shift η' is the prenexation of η via the linearization f_1 . Note that the mapping f that would produce η'' is not a linearization, since that would violate Theorem 1. In particular, $u_1 \leq v_1$ but $f(v_1) < f(u_1)$.

4.2 Alternation Height of Quantrees

So far we have not shown that linearizations even exist. Given the theoretical and empirical impact of the number of quantifier alternations on QBF solving, we are not just interested in their existence, but rather on linearizations that minimize the maximum rank N . We will now show how to compute the minimal value of N for which linearizations exist; in fact, this value will be extremely useful to extend the ideas from [6] to arbitrary QBFs in Sect. 5.2.

Consider an arbitrary quantree T , and a path x_1, \dots, x_n in T . We call this path *alternating* whenever $q(x_i) \neq q(x_{i+1})$ for $1 \leq i < n$. Then we can define the *alternation height* of T as

$$\text{aht}(T) = \max\{n \geq 0 \mid \text{there is an alternating path } x_1, \dots, x_n \text{ in } T\}.$$

Intuitively, the alternation height of T is the height of the tree that results from “clumping” together all adjacent nodes with the same quantifier. It then becomes apparent that any linearization f over T must have $N \geq \text{aht}(T)$, since for any alternating path x_1, \dots, x_n we have $f(x_1) < \dots < f(x_n)$. The following result shows that this lower bound can indeed be realized:

Theorem 2. *Let T be a quantree. Then, a linearization $f : T \rightarrow \{1, \dots, \text{aht}(T)\}$ exists. Furthermore, there exists no linearization $g : T \rightarrow \{1, \dots, N\}$ such that $0 \leq N < \text{aht}(T)$.*

Observe that the number of quantifier alternations in a prenexation via a linearization grows with the value N . In the following sections, we will restrict our scope to linearizations that minimize this value, i.e. linearizations in the set

$$\text{Lin}(T) = \{f : T \rightarrow \{1, \dots, N\} \mid f \text{ is a linearization and } N = \text{aht}(T)\}.$$

5 Linearization Strategies

We now follow up on the ideas from [6] and formalize them. In particular, we aim to obtain a formal definition of when does a linearization follow a given strategy, to ascertain whether strategies determine a unique linearization for each quantree, and to find simple algorithms to compute this linearization.

5.1 Strategies as Preferences over Linearizations

Here we take a non-constructive approach. For each prenexing strategy, we define a preference relation between the linearizations in $\text{Lin}(T)$; linearizations that follow the strategy “better” than others are preferred. As we will show, this induces a strategy-based partial order between linearizations. The desired output of a strategy must then be a maximal element w.r.t. this partial order.

The strategies from [6] are based on the idea of pushing quantifiers of a given polarity as high or as low as possible in the quantifier hierarchy. This lends itself to a natural definition of preference.

Consider an arbitrary quantree T . Given a direction $\dagger \in \{\uparrow, \downarrow\}$ and a quantifier Q , we define the *semi-preference* relation $\lesssim^{Q\dagger}$ over $\text{Lin}(T)$ given by $f \lesssim^{Q\dagger} g$ iff $f(x) \leq^\dagger g(x)$ for all $x \in T$ with $q(x) = Q$. In other words, g is preferred over f whenever g assigns ranks to Q -nodes further in the direction \dagger than f does.

Example 4. Consider the linearizations f_1, f_2 and f_6 from Fig. 1. All universal variables are assigned lower ranks by both f_1 and f_2 than by f_6 , so $f_6 \lesssim^{\forall\downarrow} f_1$ and $f_6 \lesssim^{\forall\downarrow} f_2$ hold. Note also that f_1 and f_2 assign the same ranks to universal variables, so both $f_1 \lesssim^{\forall\downarrow} f_2$ and $f_2 \lesssim^{\forall\downarrow} f_1$ hold. Note that this does not imply $f_1 = f_2$.

Example 4 shows that the antisymmetric property of partial orders does not hold for $\preceq^{Q\uparrow}$. This is intuitive: two linearizations might be just as good as each other regarding Q-nodes, while wildly differing for other nodes. Hence, for strategies to uniquely determine linearizations we need to provide preferences for both quantifiers. While this was proposed by [6], it was also noted there that uniqueness is not attained.

Example 5. The linearizations f_2 and f_3 from Fig. 1 are both good candidates for linearizations for the strategy $\exists^{\downarrow}\forall^{\uparrow}$ from [6]: both assign high ranks to existential variables and low ranks to universal variables. However, there is no apparent criterion why f_2 should or should not be preferred to f_3 under that strategy.

We solve this problem by giving one quantifier priority over the other. Our strategies are of the form $Q\uparrow\ddagger$, where Q is a quantifier and \uparrow, \ddagger are directions. A good linearization under this strategy pushes quantifiers Q in the \uparrow direction, and quantifiers \bar{Q} in the \ddagger direction; when in conflict, the former should prevail.

To formalize this idea, we liberally borrow from the somewhat similar notion of lexicographic orderings. Let us define $f \approx^Q g$ whenever $f(x) = g(x)$ for all $x \in T$ with $g(x) = Q$. In other words, $f \approx^Q g$ holds whenever both $f \preceq^{Q\uparrow} g$ and $g \preceq^{Q\uparrow} f$ hold, regardless of the choice of direction \uparrow . We define the preference relation $\preceq^{Q\uparrow\ddagger}$ over $\text{Lin}(T)$ given by $f \preceq^{Q\uparrow\ddagger} g$ iff $f \preceq^{Q\uparrow} g$ holds, and whenever $f \approx^Q g$ holds then $f \preceq^{\bar{Q}\ddagger} g$ holds as well.

Proposition 3. $\preceq^{Q\uparrow\ddagger}$ defines a partial order on $\text{Lin}(T)$.

It is easy to check that $\preceq^{\forall\uparrow\uparrow}$ and $\preceq^{\exists\uparrow\uparrow}$ are the same preference relation for $\uparrow \in \{\uparrow, \downarrow\}$. Hence, our approach defines 6 unique strategies, while [6] only proposed 4 strategies. On the one hand, the $\exists^{\uparrow}\forall^{\uparrow}$ strategy from [6] corresponds to our $\exists\uparrow\uparrow$ (or, equivalently, $\forall\uparrow\uparrow$) strategy. On the other hand, our strategies $\exists\uparrow\ddagger$ and $\forall\uparrow\ddagger$ with $\uparrow \neq \ddagger$ are both covered by the $\exists^{\uparrow}\forall^{\ddagger}$ strategy from [6], which is not uniquely determined.

Example 6. Although we cannot yet convince the reader of this, the linearizations given in Fig. 1 are the maximum element of $\text{Lin}(T)$ for each of the six (unique) preference orderings $\preceq^{Q\uparrow\ddagger}$; the corresponding strategy is shown in the rightmost column. For now, we can foreshadow that strategies $Q\uparrow\uparrow$ and $Q\uparrow\downarrow$ assign the same ranks to Q-nodes. As shown later in Lemma 1, this holds in general.

5.2 Optimal Linearizations over a Strategy

Proposition 3 suggests this is a good direction to formalize the idea of strategies: since $\text{Lin}(T)$ is finite, there exist optimal linearizations w.r.t. the preference ordering $\preceq^{Q\uparrow\ddagger}$. We call these linearizations $Q\uparrow\ddagger$ -optimal; linearizing a quantree T through the strategy $Q\uparrow\ddagger$ then means computing a $Q\uparrow\ddagger$ -optimal linearization in $\text{Lin}(T)$.

Some hurdles remain unsolved, though. For one, we have not determined if $Q\uparrow\ddagger$ -optimal linearizations are unique (i.e. if maximal elements w.r.t. $\preceq^{Q\uparrow\ddagger}$ are maxima as well). This is of interest because to empirically test the performance effect of quantifier shifting strategies, the outcome of applying a strategy must be reproducible at worst, and uniquely determined by definition at best. A second issue is a consequence of our non-constructive approach: we are yet to provide a procedure that computes a $Q\uparrow\ddagger$ -optimal linearization for a given quantree.

The rest of this section is devoted to computing a closed-form expression for $Q\uparrow\ddagger$ -optimal linearizations. Since this expression is deterministic and computable, this solves both aforementioned issues.

Overview. As we mentioned above, $\preceq^{Q\uparrow\ddagger}$ is somewhat similar to a lexicographic ordering in two components where the first component is ordered by $\preceq^{Q\uparrow}$ and the second component is ordered by $\preceq^{\overline{Q}\ddagger}$. We exploit this intuition to construct $Q\uparrow\ddagger$ -optimal linearizations: we will first optimize the first component (in our case, the ranks of Q -nodes), and then optimize the second component (the ranks of \overline{Q} -nodes) while keeping the first component fixed.

To optimize the first component, we find a linearization Γ_{\ddagger} , defined below in (1), that is optimal for $\preceq^{Q\uparrow}$. This is more precisely expressed in Lemma 1: Γ_{\ddagger} pushes Q -nodes further in direction \ddagger than any other linearization.

Interestingly, Γ_{\ddagger} does not depend on Q : Γ_{\ddagger} actually optimizes *all* nodes in the \ddagger direction. The second part of our method optimizes the ranks assigned to \overline{Q} -nodes in the \ddagger direction while keeping Q -nodes constant. For a general linearization f , this procedure results in a new linearization $[f]^{Q\ddagger}$ defined below in (2). Lemma 2 shows that $[f]^{Q\ddagger}$ is optimal for $\preceq^{Q\uparrow\ddagger}$ among the linearizations that assign the same ranks as f to Q -nodes. These two results are combined in Theorem 3: the unique $Q\uparrow\ddagger$ -maximal linearization is $[\Gamma_{\ddagger}]^{Q\ddagger}$.

Theoretical Results. Let us consider a quantree (T, \leq, q) and a strategy $Q\uparrow\ddagger$. We define the mapping $\Gamma_{\ddagger} : T \rightarrow \{1, \dots, \text{aht}(T)\}$ given by

$$\Gamma_{\ddagger}(x) = \lfloor \max^{\ddagger} \{1, \dots, \text{aht}(T)\} - \text{aht}(T_x^{\ddagger}) \rfloor + 1 \Big|_{q^*(x)}^{\ddagger}. \tag{1}$$

Furthermore, we define the mapping $[f]^{Q\ddagger} : T \rightarrow \{1, \dots, \text{aht}(T)\}$ for $f \in \text{Lin}(T)$ given by

$$[f]^{Q\ddagger}(x) = \lfloor \min^{\ddagger} \{f(y) \mid y \in T_x^{\ddagger} \text{ and } q(y) = Q\} \rfloor \Big|_{q^*(x)}^{\ddagger}. \tag{2}$$

In (2), \min^{\ddagger} is taken over a subset of $\{1, \dots, \text{aht}(T)\}$; we follow the convention that $\min^{\ddagger}(\emptyset) = \max^{\ddagger} \{1, \dots, \text{aht}(T)\}$.

Lemma 1. $\Gamma_{\ddagger} \in \text{Lin}(T)$. Furthermore, for any $g \in \text{Lin}(T)$, we have $g \preceq^{Q\uparrow} \Gamma_{\ddagger}$.

Lemma 2. $[f]^{Q\ddagger} \in \text{Lin}(T)$ for all $f \in \text{Lin}(T)$. Furthermore, $[f]^{Q\ddagger} \approx^Q f$, and for any $g \in \text{Lin}(T)$ with $g \approx^Q f$, we have $g \preceq^{Q\uparrow\ddagger} [f]^{Q\ddagger}$.

Theorem 3. Let $f \in \text{Lin}(T)$ be a $Q\uparrow\ddagger$ -optimal linearization. Then, $f = [\Gamma_{\ddagger}]^{Q\ddagger}$. In particular, $[\Gamma_{\ddagger}]^{Q\ddagger}$ is the maximum element in $(\text{Lin}(T), \preceq^{Q\uparrow\ddagger})$.

Example 7. Let us check that $[\Gamma_{\downarrow}]^{\exists\uparrow}$ is indeed f_3 for the quantree in Fig. 1 for a few values. First note that $[\Gamma_{\downarrow}]^{\exists\uparrow}$ only depends on the values of Γ_{\downarrow} for existential nodes, so we only need to compute these. In this case, $\max^{\dagger}\{1, \dots, \text{aht}(T)\} = \text{aht}(T) = 5$, $q^*(x) = 1$, and $\text{aht}(T_x^{\downarrow}) = \text{aht}(T_x)$ is simply the maximum number of quantifier alternations below x . Γ_{\downarrow} respects the tree ordering, so we obtain

$$[\Gamma_{\downarrow}]^{\exists\uparrow}(z_1) = \Gamma_{\downarrow}(z_1) = \lfloor 5 - \text{aht}(T_{z_1}) \rfloor + 1 \rfloor_1 = \lfloor 4 \rfloor_1 = 3 = f_3(z_1).$$

Furthermore, we can compute $[\Gamma_{\downarrow}]^{\exists\uparrow}(u_1)$ by checking only $\Gamma_{\downarrow}(z_1)$, since z_1 realizes the \min^{\uparrow} operator in (2). Then,

$$[\Gamma_{\downarrow}]^{\exists\uparrow}(u_1) = [\Gamma_{\downarrow}(z_1)]_{q^*(u_1)} = \lfloor 3 \rfloor_0 = 4 = f_3(u_1).$$

Example 7 suggests that $[f]^{\text{Q}\ddagger}$ can be computed recursively. Indeed, the rank of a node can be computed based on the ranks of its children or parent.

Corollary 1. *Let $x \in T$ such that $q(x) \neq \text{Q}$. Then,*

$$[f]^{\text{Q}\ddagger}(x) = \left[\min^{\ddagger} \{ [f]^{\text{Q}\ddagger}(y) \mid x \text{ is covered by } y \in T \text{ w.r.t. } \leq^{\ddagger} \} \right]_{q^*(x)}^{\ddagger}$$

6 Implementation and Evaluation

We implemented the optimal linearization $[\Gamma_{\ddagger}]^{\text{Q}\ddagger}$ for each strategy $\text{Q}\ddagger$ described in Sect. 5. Our implementation uses the BOOLEGURU framework [10], designed for efficiently working with propositional formulas and QBFs. BOOLEGURU provides a convenient parsing and serialization infrastructure for widely used formats, as well as helper functions to write formula transformations. Our extension is licensed under the MIT license and publicly available².

Our implementation computes a quantifier shift on an input QBF φ based on a strategy $\text{Q}\ddagger$ by traversing twice the abstract syntax tree of the parsed QBF φ in a depth-first fashion. In the first pass, the propositional skeleton φ_{psk} and the quantree T are extracted. Furthermore, the values $\text{aht}(T^x)$ and $\text{aht}(T_x)$, which we call *height* and *depth* of x , are computed for each node $x \in T$.

The second pass is applied only to the quantree. For each node $x \in T$, we compute its rank $[\Gamma_{\ddagger}]^{\text{Q}\ddagger}(x)$. For Q-nodes, this rank is given by $\Gamma_{\ddagger}(x)$, which is trivial to compute from the height and depth of x ; for $\bar{\text{Q}}$ -nodes, Corollary 1 allows a recursive computation. Based on their rank, quantifier nodes in the quantree are collected in a quantifier prefix which is appended to φ_{psk} .

To apply a linearization strategy to an arbitrary formula, BOOLEGURU needs to be called with the options `:linearize-quant- $\{\text{E}, \text{A}\}$ - $\{\text{up}, \text{down}\}$ - $\{\text{up}, \text{down}\}$` using the quantifier E (\exists) or A (\forall) and the two directions `up` (\uparrow) and `down` (\downarrow). Overall, there are eight different combinations that we evaluate in the following. However, from the discussion above, it becomes obvious that only six of

² <https://github.com/maximaximal/booleguru>.

those eight strategies are different. In the implementation, all quantifiers are first extracted from an expression and processed in a separate tree. Each node contains the quantifier type, the quantified variables, and dependent quantifier nodes.

After computing the linearization, the extracted quantifiers are inserted piecewise as new expressions that wrap the originally transformed expression. This ensures the ordering of variables within the quantifier blocks stays the same. The fully quantified expression is then returned from the transformer and can either be printed using one of BOOLEGURU's serializers, or processed further.

6.1 Benchmarks

As most solvers only process formulas in prenex (conjunctive) normal form, hardly any non-prenex benchmarks are currently available. To test our implementation, we considered the benchmark set from the QBFEval 2008 and we reimplemented a generator for nested counterfactuals as described below. All used formulas and corresponding experimental logs are available at [11].

Nested Counterfactuals. We developed a novel generator for nested counterfactuals (NCFs) based on a Lua script, which is integrated into BOOLEGURU. The full encoding is described in [6]. To generate NCFs, five arguments must be provided: numbers of formulas in the background theory, numbers of variables, clauses per formula, variables per clause, nesting depth. Optionally, a sixth argument to fix the seed value for random choices. A counterfactual $\phi > \psi$ is true over a background theory T iff the minimal change of T to incorporate ϕ entails ψ . In a nested counterfactual, also ϕ or ψ are allowed to be (nested) counterfactuals. For details see [6]. We chose the range of arguments based on the description mentioned in Egly et al. [6]. We assume that the background theory T always consists of 5 randomly generated formulas. Each of these formulas consists of 2 to 10 clauses where each clause is a disjunction of 3 variables. The clauses contain randomly chosen atoms from a set of 5 variables. These atoms have a 50 percent chance of being negated. No clause may contain the same literal more than once and the clauses are non-tautological. The nesting depth of the counterfactuals ranges from 2 to 6. All possible combinations of these selected parameters result in 45 different classes. For each of these classes, 100 instances were generated to ensure that both, satisfiable and unsatisfiable results are represented. With the 8 strategies we obtain 36 000 prenexed formulas either in the non-CNF QCIR format or in QDIMACS.

Non-Prenex-Non-CNF Benchmarks from QBFEval 2008. In the QBFEval 2008, a non-prenex, non-CNF track was organized [19]. The benchmarks are available at the QBFLib.³ This set consists of 492 formulas in the outdated Boole format. To transform these formulas into prenex form, we first rewrote them into

³ <http://www.qbflib.org>.

Table 1. Number of solved formulas per strategy and solver of QBFEval’08 set (QCIR). Diff indicates the difference between the best and the worst strategy. Each strategy has 492 formulas.

Solver	$\exists\uparrow\uparrow$	$\exists\downarrow\downarrow$	$\exists\uparrow\downarrow$	$\exists\downarrow\uparrow$	$\forall\uparrow\uparrow$	$\forall\downarrow\downarrow$	$\forall\uparrow\downarrow$	$\forall\downarrow\uparrow$	Diff.	Rel. diff. (%)
QUABS	380	398	405	366	380	400	376	406	40	8.13
QFUN	340	409	360	339	340	407	341	361	70	14.23
CQUESTO	436	451	430	442	436	450	447	429	22	4.47
QUTE	455	464	465	452	455	464	454	465	13	2.64

the related Limboole⁴ format that is processable by BOOLEGURU. Again, we considered all eight options resulting in 4936 prenexed formulas.

6.2 Experimental Setup

All experiments were run with a timeout of 15 minutes on a cluster of dual-socket AMD EPYC 7313 @ 3.7GHz machines running Ubuntu 22.04 with a 8GB memory limit per task. We split the experiments into two parts: on the one hand, we consider solvers that process formulas in prenex conjunctive normal form (PCNF) and on the other hand, we consider solvers that process formulas in prenex non-CNF. For the first group of solvers that accept formulas in the QDIMACS format, we consider the following solvers: The solver DEPQBF (version 6.03) is a conflict/solution driven clause/cube learning (QCDCL) solver that integrates several advanced inprocessing techniques and reasoning under the standard dependency scheme [17]. Also QUTE [21] is a QCDCL solver that employs dynamic dependency learning. This solver is also able to process QCIR formulas, i.e., it is also included in the second group. The solver CAQE [25] (version 4.0.2) implements clausal selection. The solver RAREQS [14] (version 1.1) implements variable expansion in CEGAR style. Finally, DYNQBF [4, 5] (version 1.1.1) is a BDD-based solver. For pre-processing, we used BLOQQER [3] and HQSPRE [31]. For testing the encodings in the non-CNF QCIR format, we include the solvers QUABS [9, 29] and CQUESTO [13] (version v00.0) (*sic*) that lift clausal selection to circuits, QFUN [12] (version v00.0) (*sic*), a solver that employs machine learning techniques, and QUTE which was already mentioned above.

6.3 Experimental Results

In the following, we first discuss the results of the solvers that process formulas in QCIR, (i.e. formulas in prenex form but not in CNF). Second, we report on our experiments with QDIMACS formulas for the PCNF solvers.

⁴ <http://fmv.jku.at/limboole/>.

Table 2. Number of different prefixes generated from of the 2008 non-CNF benchmark set with all strategy combinations. Each strategy has 492 formulas.

$\forall \backslash \exists$	$\downarrow\downarrow$	$\downarrow\uparrow$	$\uparrow\downarrow$	$\uparrow\uparrow$
$\downarrow\downarrow$	0	4500	4500	4500
$\downarrow\uparrow$	4500	4500	0	4500
$\uparrow\downarrow$	4500	0	4500	4500
$\uparrow\uparrow$	4500	4500	4500	0

Prenexed formulas in QCIR. The nested counterfactual benchmarks were easily solved by QCIR solvers, i.e., they could exploit the formula structure to quickly solve these formulas (all were solved in less than a second). Therefore, we focus on the formulas of the QBFEval’08 benchmark set in the following. Table 1 shows the results for the QCIR solvers and Table 2 shows the number of different prefixes that were generated with all strategy combinations. For QUABS, QFUN, and CQUESTO we see a clear difference between the best and worst shifting strategy. In contrast, QUTE seems to be less sensitive regarding the prenexing, which might be related to its dynamic dependency learning. The detailed solving behavior of QFUN and CQUESTO is shown in Fig. 2. For QFUN we observe that $\exists\uparrow\uparrow$ and $\forall\uparrow\uparrow$ clearly perform best, while $\forall\downarrow\uparrow$ and $\exists\uparrow\downarrow$ seem to be less beneficial.

Prenexed Formulas in QDIMACS. Table 3 shows the results of the QDIMACS solvers on the encodings of the nested counterfactuals and Table 4 shows the number of different prefixes that were generated between all strategy combinations. DEPQBF solves all formulas from 4 of the 8 strategies and most of the others, DYNQBF is able to solve most of the formulas and QUTE solves about one quarter of the formulas. Meanwhile RReQS and CAQE hardly solve any of those. These could be connected with the observation that those solvers perform better on formulas with few quantifier alternations. For all solvers we observe

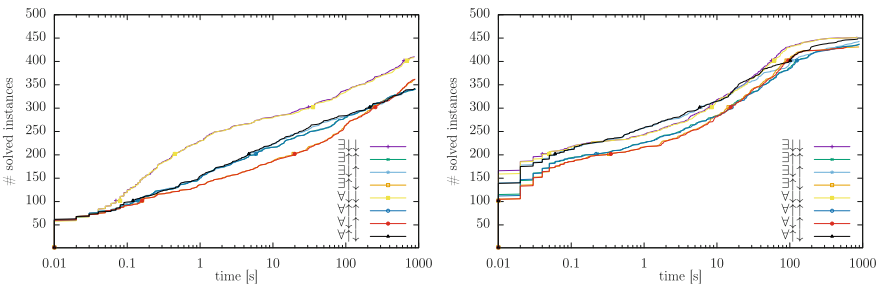


Fig. 2. Solving time of the QBFEVAL’08 set with QFUN (left) and CQUESTO (right).

Table 3. Number of solved formulas per strategy and solver of NCFs. Diff indicates the difference between the best and the worst strategy. Each strategy has 4500 formulas.

Solver	$\exists\uparrow\uparrow$	$\exists\downarrow\downarrow$	$\exists\uparrow\downarrow$	$\exists\downarrow\uparrow$	$\forall\uparrow\uparrow$	$\forall\downarrow\downarrow$	$\forall\uparrow\downarrow$	$\forall\downarrow\uparrow$	Diff.	Rel. diff. (%)
DEPQBF	4500	4495	4497	4500	4500	4495	4500	4497	5	0.11
CAQE	37	86	88	37	37	86	37	88	51	1.13
RAReQS	21	12	19	16	21	12	16	20	9	0.2
QUTE	1012	731	724	1010	1012	731	1010	724	288	6.4
DYNQBF	4274	4456	4318	4467	4279	4469	4474	4316	200	4.44

Table 4. Number of different prefixes generated from of the NCF benchmark set with all strategy combinations. Each strategy has 4500 formulas.

$\backslash \exists$	$\downarrow\downarrow$	$\downarrow\uparrow$	$\uparrow\downarrow$	$\uparrow\uparrow$
\forall				
$\downarrow\downarrow$	0	4500	4500	4500
$\downarrow\uparrow$	4500	4500	0	4500
$\uparrow\downarrow$	4500	0	4500	4500
$\uparrow\uparrow$	4500	4500	4500	0

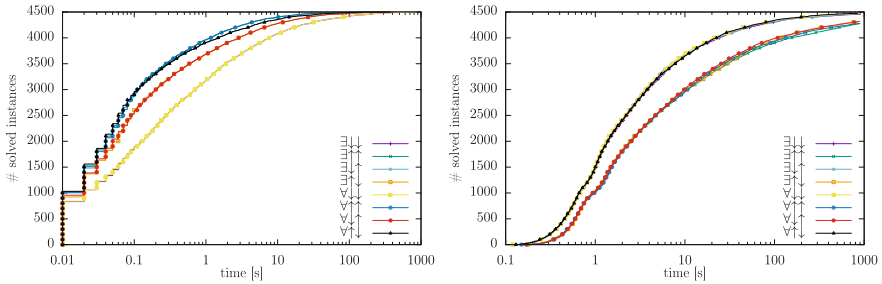


Fig. 3. Solving time of nested counterfactuals with DEPQBF (left) and DYNQBF (right).

that the chosen shifting strategy impacts the number of the solved formulas. Details of the runs of DEPQBF and DYNQBF are shown in Fig. 3. For DEPQBF, we observe that strategies $\exists\downarrow\downarrow$ and $\forall\downarrow\downarrow$ are clearly less preferable than strategies $\exists\uparrow\uparrow$ and $\forall\uparrow\uparrow$, while DYNQBF prefers to have existential quantifiers shifted down. The QBFEval’08 benchmarks are very challenging for recent QDIMACS solvers with our encoding. Out of the 492 formulas, DEPQBF solves up to 128 formulas with the best strategy ($\forall\downarrow\downarrow$). DYNQBF solves around 60 formulas. The other tools solve less than 30 formulas. Enabling preprocessing is beneficial for all solvers. When preprocessors BLOQER or HQSPRE simplify the formulas, then almost all formulas can be solved. With and without preprocessing, the

shifting strategies have only little impact on this benchmark set. Note that more than two third of these formulas have five or less quantifier alternations.

7 Conclusion and Future Work

This paper analyzes and extends previous work from 2003 on quantifier shifting for quantified Boolean formulas. Since then, much progress has been made in the development of QBF solvers by introducing novel solving paradigms, applying efficient preprocessing techniques, and exploiting quantifier (in-)dependence. However, most of those approaches assume formulas in prenex normal form. As a consequence, most encodings are provided in this form, which unnecessarily restricts solvers with a certain design choice. In this work, we not only formalized prenexing in a concise manner, but we also provide an efficient, publicly available tool that implements the discussed prenexing strategies and Tseitin transformation. In extensive experiments with state-of-the-art prenex CNF and non-CNF solvers, we showed that in many instances prenexing strategy selection impacts solving runtime. We showed that different solvers perform differently on different strategies, hence it was not possible to uniquely identify the best strategy. Therefore, we think it is important that solver developers and also the developers of QBF encodings exploit information available in the problem structure and do not introduce artificial restrictions.

In future work, we plan to design and evaluate further prenexing strategies and we will also revisit more non-prenex QBF encodings to obtain larger benchmark sets. At the moment, hardly any formulas in non-prenex form are available which we changed by providing the generator for encodings of nested counterfactuals. But this is a first step only. Many of the considered formulas are either too hard or too easy for recent solvers, hence more effort is necessary to obtain a larger variety of interesting benchmarks (also in the light of next QBF evaluations). Finally, we want to explore how prenexing strategies affect the generation of certificates and solutions in terms of Herbrand and Skolem functions. From first-order logic, it is well known that it is beneficial to move quantifiers as far inwards as possible to minimize the arity of the first-order Skolem functions [20].

References

1. Beyersdorff, O., Hinde, L., Pich, J.: Reasons for hardness in QBF proof systems. *ACM Trans. Comput. Theory* **12**(2), 10:1–10:27 (2020). <https://doi.org/10.1145/3378665>
2. Beyersdorff, O., Janota, M., Lonsing, F., Seidl, M.: Quantified Boolean formulas. In: *Handbook of Satisfiability – Second Edition, Frontiers in Artificial Intelligence and Applications*, vol. 336, pp. 1177–1221. IOS Press (2021). <https://doi.org/10.3233/FAIA201015>

3. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 101–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_10
4. Charwat, G., Woltran, S.: BDD-based dynamic programming on tree decompositions. Technical report, Technische Universität Wien, Institut für Informationssysteme, Technical report (2016)
5. Charwat, G., Woltran, S.: Dynamic programming-based QBF solving. In: Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016) co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016), Bordeaux, France, July 4, 2016. CEUR Workshop Proceedings, vol. 1719, pp. 27–40. CEUR-WS.org (2016)
6. Egly, U., Seidl, M., Tompits, H., Woltran, S., Zolda, M.: Comparing different prenexing strategies for quantified Boolean formulas. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 214–228. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_17
7. Egly, U., Seidl, M., Woltran, S.: A solver for QBFS in nonprenex form. In: ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 477–481. IOS Press (2006)
8. Goultiaeva, A., Bacchus, F.: Exploiting circuit representations in QBF solving. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 333–339. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_29
9. Hecking-Harbusch, J., Tentrup, L.: Solving QBF by abstraction. In: Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26–28th September 2018. EPTCS, vol. 277, pp. 88–102 (2018). <https://doi.org/10.4204/EPTCS.277.7>
10. Heisinger, M., Heisinger, S., Seidl, M.: Booleguru, the propositional polyglot. In: Benz Müller, C., Heule, M., Schmidt, R. (eds.) Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3–6, 2024, Proceedings. LNCS, vol. 14739, p. 315–324. Springer (2024). https://doi.org/10.1007/978-3-031-63498-7_19
11. Heisinger, S., Heisinger, M., Rebola-Pardo, A., Seidl, M.: Artifact for “quantifier shifting for quantified Boolean formulas revisited” (2024). <https://doi.org/10.5281/zenodo.10634925>
12. Janota, M.: QFUN: towards machine learning in QBF. CoRR **abs/1710.02198** (2017)
13. Janota, M.: Circuit-based search space pruning in QBF. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 187–198. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_12
14. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_10
15. Klieber, W., Sapa, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF Solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_12

16. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 158–171. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_14
17. Lonsing, F., Egly, U.: DepQBF 6.0: a search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 371–384. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_23
18. Lonsing, F., Egly, U.: Evaluating QBF solvers: quantifier alternations matter. In: Hooker, J. (ed.) CP 2018. LNCS, vol. 11008, pp. 276–294. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98334-9_19
19. Marin, P., Narizzano, M., Pulina, L., Tacchella, A., Giunchiglia, E.: Twelve years of QBF evaluations: QSAT is PSPACE-Hard and it shows. *Fundam. Informaticae* **149**(1–2), 133–158 (2016). <https://doi.org/10.3233/FI-2016-1445>
20. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Handbook of Automated Reasoning (in 2 volumes), pp. 335–367. Elsevier and MIT Press (2001). <https://doi.org/10.1016/B978-044450813-3/50008-4>
21. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 298–313. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_19
22. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. *J. Artif. Intell. Res.* **65**, 180–208 (2019). <https://doi.org/10.1613/jair.1.11529>
23. Peterson, G., Reif, J., Azhar, S.: Lower bounds for multiplayer noncooperative games of incomplete information. *Comput. Math. App.* **41**(7–8), 957–992 (2001)
24. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. *J. Symb. Comput.* **2**(3), 293–304 (1986). [https://doi.org/10.1016/S0747-7171\(86\)80028-1](https://doi.org/10.1016/S0747-7171(86)80028-1)
25. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27–30, 2015. pp. 136–143. IEEE (2015). <https://doi.org/10.1109/FMCAD.2015.7542263>
26. Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Moving definition variables in quantified Boolean formulas. In: TACAS 2022. LNCS, vol. 13243, pp. 462–479. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_26
27. Samer, M., Szeider, S.: Backdoor sets of quantified Boolean formulas. *J. Autom. Reason.* **42**(1), 77–97 (2009). <https://doi.org/10.1007/s10817-008-9114-5>
28. Shukla, A., Biere, A., Pulina, L., Seidl, M.: A survey on applications of quantified Boolean formulas. In: 31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4–6, 2019, pp. 78–84. IEEE (2019). <https://doi.org/10.1109/ICTAI.2019.00020>
29. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 393–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_24
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. Symbolic Computation, pp. 466–483. Springer, Berlin, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28
31. Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre – an effective preprocessor for QBF and DQBF. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 373–390. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_21

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

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





Satisfiability Modulo Exponential Integer Arithmetic

Florian Frohn¹ and Jürgen Giesl¹

RWTH Aachen University, Aachen, Germany
{florian.frohn,giesl}@informatik.rwth-aachen.de

Abstract. SMT solvers use sophisticated techniques for polynomial (linear or non-linear) integer arithmetic. In contrast, non-polynomial integer arithmetic has mostly been neglected so far. However, in the context of program verification, polynomials are often insufficient to capture the behavior of the analyzed system without resorting to approximations. In the last years, *incremental linearization* has been applied successfully to satisfiability modulo real arithmetic with transcendental functions. We adapt this approach to an extension of polynomial integer arithmetic with exponential functions. Here, the key challenge is to compute suitable *lemmas* that eliminate the current model from the search space if it violates the semantics of exponentiation. An empirical evaluation of our implementation shows that our approach is highly effective in practice.

1 Introduction

Traditionally, automated reasoning techniques for integers focus on polynomial arithmetic. This is not only true in the context of SMT, but also for program verification techniques, since the latter often search for polynomial invariants that imply the desired properties. As invariants are over-approximations, they are well suited for proving “universal” properties like safety, termination, or upper bounds on the worst-case runtime that refer to all possible program runs. However, proving dual properties like unsafety, non-termination, or lower bounds requires under-approximations, so that invariants are of limited use here.

For lower bounds, an *infinite set* of witnesses is required, as the runtime w.r.t. a finite set of (terminating) program runs is always bounded by a constant. Thus, to prove non-constant lower bounds, *symbolic under-approximations* are required, i.e., formulas that describe an infinite subset of the reachable states. However, polynomial arithmetic is often insufficient to express such approximations. To see this, consider the program

```
 $x \leftarrow 1; y \leftarrow \text{nondet}(0, \infty); \text{while } y > 0 \text{ do } x \leftarrow 3 \cdot x; y \leftarrow y - 1 \text{ done}$ 
```

where $\text{nondet}(0, \infty)$ returns a natural number non-deterministically. Here, the set of reachable states after execution of the loop is characterized by the formula

$$\exists n \in \mathbb{N}. x = 3^n \wedge y = 0. \tag{1}$$

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2).

In recent work, *acceleration techniques* have successfully been used to deduce lower runtime bounds automatically [17, 18]. While they can easily derive a formula like (1) from the code above, this is of limited use, as most¹ SMT solvers cannot handle terms of the form 3^n . Besides lower bounds, acceleration has also successfully been used for proving non-termination [15, 18, 19] and (un)safety [3, 6, 7, 20, 28, 29], where its strength is finding long counterexamples that are challenging for other techniques.

Importantly, exponentiation is not just “yet another function” that can result from applying acceleration techniques. There are well-known, important classes of loops where polynomials and exponentiation *always* suffice to represent the values of the program variables after executing a loop [16, 26]. Thus, the lack of support for integer exponentiation in SMT solvers is a major obstacle for the further development of acceleration-based verification techniques.

In this work, we first define a novel SMT theory for integer arithmetic with exponentiation. Then we show how to lift standard SMT solvers to this new theory, resulting in our novel tool `SwlnE` (`SMT with Integer Exponentiation`).

Our technique is inspired by *incremental linearization*, which has been applied successfully to *real* arithmetic with transcendental functions, including the natural exponential function $\exp_e(x) = e^x$, where e is Euler’s number [11]. In this setting, incremental linearization considers \exp_e as an uninterpreted function. If the resulting SMT problem is unsatisfiable, then so is the original problem. If it is satisfiable and the model that was found for \exp_e coincides with the semantics of exponentiation, then the original problem is satisfiable. Otherwise, *lemmas* about \exp_e that rule out the current model are added to the SMT problem, and then its satisfiability is checked again. The name “incremental linearization” is due to the fact that these lemmas only contain linear arithmetic.

The main challenge for adapting this approach to integer exponentiation is to generate suitable lemmas, see Sect. 4.2. Except for so-called *monotonicity lemmas*, none of the lemmas from [11] easily carry over to our setting. In contrast to [11], we do not restrict ourselves to linear lemmas, but we also use non-linear, polynomial lemmas. This is due to the fact that we consider a binary version $\lambda x, y. x^y$ of exponentiation, whereas [11] fixes the base to e . Thus, in our setting, one obtains *bilinear* lemmas that are linear w.r.t. x as well as y , but may contain multiplication between x and y (i.e., they may contain the subterm $x \cdot y$). More precisely, bilinear lemmas arise from *bilinear interpolation*, which is a crucial ingredient of our approach, as it allows us to eliminate *any* model that violates the semantics of exponentiation (Theorem 23). Therefore, the name “incremental linearization” does not fit to our approach, which is rather an instance of “counterexample-guided abstraction refinement” (CEGAR) [13].

To summarize, our contributions are as follows: We first propose the new SMT theory EIA for integer arithmetic with exponentiation (Sect. 3). Then, based on novel techniques for generating suitable lemmas, we develop a CEGAR approach for EIA (Sect. 4). We implemented our approach in our novel open-

¹ CVC5 uses a dedicated solver for [integer exponentiation with base 2](#).

source tool SwInE [22,23] and evaluated it on a collection of 4627 EIA benchmarks that we synthesized from verification problems. Our experiments show that our approach is highly effective in practice (Sect.6). All proofs can be found in [21].

2 Preliminaries

We are working in the setting of *SMT-LIB logic* [4], a variant of many-sorted first-order logic with equality. We now introduce a reduced variant of [4], where we only explain those concepts that are relevant for our work.

In SMT-LIB logic, there is a dedicated Boolean sort **Bool**, and hence formulas are just terms of sort **Bool**. Similarly, there is no distinction between predicates and functions, as predicates are simply functions of type **Bool**.

So in SMT-LIB logic, a *signature* $\Sigma = (\Sigma^S, \Sigma^F, \Sigma^R)$ consists of a set Σ^S of *sorts*, a set Σ^F of *function symbols*, and a *ranking function* $\Sigma^R : \Sigma^F \rightarrow (\Sigma^S)^+$. The meaning of $\Sigma^R(f) = (s_1, \dots, s_k)$ is that f is a function which maps arguments of the sorts s_1, \dots, s_{k-1} to a result of sort s_k . We write $f : s_1 \dots s_k$ instead of “ $f \in \Sigma^F$ and $\Sigma^R(f) = (s_1, \dots, s_k)$ ” if Σ is clear from the context. We always allow to implicitly extend Σ with arbitrarily many constant function symbols (i.e., function symbols x where $|\Sigma^R(x)| = 1$). Note that SMT-LIB logic only considers closed terms, i.e., terms without free variables, and we are only concerned with quantifier-free formulas, so in our setting, all formulas are ground. Therefore, we refer to these constant function symbols as *variables* to avoid confusion with other, predefined constant function symbols like **true**, **0**, \dots , see below.

Every SMT-LIB signature is an extension of $\Sigma_{\mathbf{Bool}}$ where $\Sigma_{\mathbf{Bool}}^S = \{\mathbf{Bool}\}$ and $\Sigma_{\mathbf{Bool}}^F$ consists of the following function symbols:

$$\mathbf{true}, \mathbf{false} : \mathbf{Bool} \quad \neg : \mathbf{Bool} \mathbf{Bool} \quad \wedge, \vee, \implies, \iff : \mathbf{Bool} \mathbf{Bool} \mathbf{Bool}$$

Note that SMT-LIB logic only considers well-sorted terms. A Σ -*structure* **A** consists of a *universe* $A = \bigcup_{s \in \Sigma^S} A_s$ and an *interpretation function* that maps each function symbol $f : s_1 \dots s_k$ to a function $\llbracket f \rrbracket^{\mathbf{A}} : A_{s_1} \times \dots \times A_{s_{k-1}} \rightarrow A_{s_k}$. SMT-LIB logic only considers structures where $A_{\mathbf{Bool}} = \{\mathbf{true}, \mathbf{false}\}$ and all function symbols from $\Sigma_{\mathbf{Bool}}$ are interpreted as usual.

A Σ -*theory* is a class of Σ -structures. For example, consider the extension $\Sigma_{\mathbf{Int}}$ of $\Sigma_{\mathbf{Bool}}$ with the additional sort **Int** and the following function symbols:

$$0, 1, \dots : \mathbf{Int} \quad +, -, \cdot, \text{div}, \text{mod} : \mathbf{Int} \mathbf{Int} \mathbf{Int} \quad <, \leq, >, \geq, =, \neq : \mathbf{Int} \mathbf{Int} \mathbf{Bool}$$

Then the $\Sigma_{\mathbf{Int}}$ -theory *non-linear integer arithmetic* (NIA)² contains all $\Sigma_{\mathbf{Int}}$ -structures where $A_{\mathbf{Int}} = \mathbb{Z}$ and all symbols from $\Sigma_{\mathbf{Int}}$ are interpreted as usual.

² As we only consider quantifier-free formulas, we omit the prefix “QF.” in theory names and write, e.g., NIA instead of QF_NIA. In [4], QF_NIA is called an *SMT-LIB logic*, which restricts the (first-order) *theory* of integer arithmetic to the quantifier-free fragment. For simplicity, we do not distinguish between SMT-LIB logics and theories.

If \mathbf{A} is a Σ -structure and Σ' is a subsignature of Σ , then the *reduct* of \mathbf{A} to Σ' is the unique Σ' -structure that interprets its function symbols like \mathbf{A} . So the theory *linear integer arithmetic* (LIA) consists of the reducts of all elements of NIA to $\Sigma_{\mathbf{Int}} \setminus \{\cdot, \text{div}, \text{mod}\}$.

Given a Σ -structure \mathbf{A} and a Σ -term t , the *meaning* $\llbracket t \rrbracket^{\mathbf{A}}$ of t results from interpreting all function symbols according to \mathbf{A} . For function symbols f whose interpretation is fixed by a Σ -theory \mathcal{T} , we denote f 's interpretation by $\llbracket f \rrbracket^{\mathcal{T}}$. Given a Σ -theory \mathcal{T} , a Σ -formula φ (i.e., a Σ -term of type **Bool**) is *satisfiable* in \mathcal{T} if there is an $\mathbf{A} \in \mathcal{T}$ such that $\llbracket \varphi \rrbracket^{\mathbf{A}} = \mathbf{true}$. Then \mathbf{A} is called a *model* of φ , written $\mathbf{A} \models \varphi$. If *every* $\mathbf{A} \in \mathcal{T}$ is a model of φ , then φ is \mathcal{T} -*valid*, written $\models_{\mathcal{T}} \varphi$. We write $\psi \equiv_{\mathcal{T}} \varphi$ for $\models_{\mathcal{T}} \psi \iff \varphi$.

We sometimes also consider *uninterpreted functions*. Then the signature may not only contain the function symbols of the theory under consideration and variables, but also additional non-constant function symbols.

We write “term”, “structure”, “theory”, ... instead of “ Σ -term”, “ Σ -structure”, “ Σ -theory”, ... if Σ is irrelevant or clear from the context. Similarly, we just write “ \equiv ” and “valid” instead of “ $\equiv_{\mathcal{T}}$ ” and “ \mathcal{T} -valid” if \mathcal{T} is clear from the context. Moreover, we use unary minus and t^c (where t is a term of sort **Int** and $c \in \mathbb{N}$) as syntactic sugar, and we use infix notation for binary function symbols.

In the sequel, we use x, y, z, \dots for variables, s, t, p, q, \dots for terms of sort **Int**, φ, ψ, \dots for formulas, and a, b, c, d, \dots for integers.

3 The SMT Theory EIA

We now introduce our novel SMT theory for *exponential integer arithmetic*. To this end, we define the signature $\Sigma_{\mathbf{Int}}^{\text{exp}}$, which extends $\Sigma_{\mathbf{Int}}$ with

$$\text{exp} : \mathbf{Int} \ \mathbf{Int} \ \mathbf{Int}.$$

If the 2^{nd} argument of exp is non-negative, then its semantics is as expected, i.e., we are interested in structures \mathbf{A} such that $\llbracket \text{exp} \rrbracket^{\mathbf{A}}(c, d) = c^d$ for all $d \geq 0$. However, if the 2^{nd} argument is negative, then we have to use different semantics. The reason is that we may have $c^d \notin \mathbb{Z}$ if $d < 0$. Intuitively, exp should be a partial function, but all functions are total in SMT-LIB logic. We solve this problem by interpreting $\text{exp}(c, d)$ as $c^{|d|}$. This semantics has previously been used in the literature, and the resulting logic admits a known decidable fragment [5].

Definition 1 (EIA). *The theory exponential integer arithmetic (EIA) contains all $\Sigma_{\mathbf{Int}}^{\text{exp}}$ -structures \mathbf{A} with $\llbracket \text{exp} \rrbracket^{\mathbf{A}}(c, d) = c^{|d|}$ whose reduct to $\Sigma_{\mathbf{Int}}$ is in NIA.*

Alternatively, one could treat $\text{exp}(c, d)$ like an uninterpreted function if d is negative. Doing so would be analogous to the treatment of division by zero in SMT-LIB logic. Then, e.g., $\text{exp}(0, -1) \neq \text{exp}(0, -2)$ would be satisfied by a structure \mathbf{A} with $\llbracket \text{exp} \rrbracket^{\mathbf{A}}(c, d) = c^d$ if $d \geq 0$ and $\llbracket \text{exp} \rrbracket^{\mathbf{A}}(c, d) = d$, otherwise. However, the drawback of this approach is that important laws of exponentiation like

$$\text{exp}(\text{exp}(x, y), z) = \text{exp}(x, y \cdot z)$$

would not be valid. Thus, we focus on the semantics from Definition 1.

Algorithm 1: CEGAR for EIA

```

Input: a  $\Sigma_{\text{Int}}^{\text{exp}}$ -formula  $\varphi$ 
// Preprocessing
1 do
2    $\varphi' \leftarrow \varphi$ ;
3    $\varphi \leftarrow \text{FOLDCONSTANTS}(\varphi)$ ;
4    $\varphi \leftarrow \text{REWRITE}(\varphi)$ ;
5 while  $\varphi \neq \varphi'$ ;
// Refinement Loop
6 while there is a NIA-model  $\mathbf{A}$  of  $\varphi$  do
7   if  $\mathbf{A}$  is a counterexample then
8      $\mathcal{L} \leftarrow \emptyset$ ;
9     for  $\text{kind} \in \{\text{Symmetry, Monotonicity, Bounding, Interpolation}\}$  do
10    |  $\mathcal{L} \leftarrow \mathcal{L} \cup \text{COMPUTELEMNAS}(\varphi, \text{kind})$ ;
11    |  $\varphi \leftarrow \varphi \wedge \bigwedge \{\psi \in \mathcal{L} \mid \mathbf{A} \not\models \psi\}$ 
12  else return sat
13 return unsat

```

4 Solving EIA Problems via CEGAR

We now explain our technique for solving EIA problems, see Algorithm 1. Our goal is to (dis)prove satisfiability of φ in EIA. The loop in Line 6 is a CEGAR loop which lifts an SMT solver for NIA (which is called in Line 6) to EIA. So the *abstraction* consists of using NIA- instead of EIA-models. Hence, exp is considered to be an uninterpreted function in Line 6, i.e., the SMT solver also searches for an interpretation of exp . If the model found by the SMT solver is a *counterexample* (i.e., if $\llbracket \text{exp} \rrbracket^{\mathbf{A}}$ conflicts with $\llbracket \text{exp} \rrbracket^{\text{EIA}}$), then the formula under consideration is refined by adding suitable lemmas in Lines 9–11 and the loop is iterated again.

Definition 2 (Counterexample). *We call a NIA-model \mathbf{A} of φ a counterexample if there is a subterm $\text{exp}(s, t)$ of φ such that $\llbracket \text{exp}(s, t) \rrbracket^{\mathbf{A}} \neq (\llbracket s \rrbracket^{\mathbf{A}}) \llbracket t \rrbracket^{\mathbf{A}}$.*

In the sequel, we first discuss our preprocessings (first loop in Algorithm 1) in Sect. 4.1. Then we explain our refinement (Lines 9–11) in Sect. 4.2. Here, we first introduce the different kinds of lemmas that are used by our implementation in Sect. 4.2.1–4.2.4. If implemented naively, the number of lemmas can get quite large, so we explain how to generate lemmas *lazily* in Sect. 4.2.5. Finally, we conclude this section by stating important properties of Algorithm 1.

Example 3 (Leading Example). To illustrate our approach, we show how to prove

$$\forall x, y. |x| > 2 \wedge |y| > 2 \implies \text{exp}(\text{exp}(x, y), y) \neq \text{exp}(x, \text{exp}(y, y))$$

by encoding absolute values suitably³ and proving unsatisfiability of its negation:

$$x^2 > 4 \wedge y^2 > 4 \wedge \exp(\exp(x, y), y) = \exp(x, \exp(y, y))$$

4.1 Preprocessings

In the first loop of Algorithm 1, we preprocess φ by alternating *constant folding* (Line 3) and *rewriting* (Line 4) until a fixpoint is reached. Constant folding evaluates subexpressions without variables, where subexpressions $\exp(c, d)$ are evaluated to $c^{|d|}$, i.e., according to the semantics of EIA. Rewriting reduces the number of occurrences of \exp via the following (terminating) rewrite rules:

$$\begin{aligned} \exp(x, c) &\rightarrow x^{|c|} && \text{if } c \in \mathbb{Z} \\ \exp(\exp(x, y), z) &\rightarrow \exp(x, y \cdot z) \\ \exp(x, y) \cdot \exp(z, y) &\rightarrow \exp(x \cdot z, y) \end{aligned}$$

In particular, the 1st rule allows us to rewrite⁴ $\exp(s, 0)$ to $s^0 = 1$ and $\exp(s, 1)$ to $s^1 = s$. Note that the rule

$$\exp(x, y) \cdot \exp(x, z) \rightarrow \exp(x, y + z)$$

would be unsound, as the right-hand side would need to be $\exp(x, |y| + |z|)$ instead. We leave the question whether such a rule is beneficial to future work.

Example 4 (Preprocessing). For our leading example, applying the 2nd rewrite rule at the underlined position yields:

$$\begin{aligned} x^2 > 4 \wedge y^2 > 4 \wedge \underline{\exp(\exp(x, y), y)} &= \exp(x, \exp(y, y)) \\ \rightarrow x^2 > 4 \wedge y^2 > 4 \wedge \exp(x, y^2) &= \exp(x, \exp(y, y)) \end{aligned} \quad (2)$$

Lemma 5. *We have $\varphi \equiv_{\text{EIA}} \text{FOLDCONSTANTS}(\varphi)$ and $\varphi \equiv_{\text{EIA}} \text{REWRITE}(\varphi)$.*

4.2 Refinement

Our refinement (Lines 9–11 of Algorithm 1) is based on the four kinds of lemmas named in Line 9: *symmetry lemmas*, *monotonicity lemmas*, *bounding lemmas*, and *interpolation lemmas*. In the sequel, we explain how we compute a set \mathcal{L} of such lemmas. Then our refinement conjoins

$$\{\psi \in \mathcal{L} \mid \mathbf{A} \not\models \psi\}$$

to φ in Line 11. As our lemmas allow us to eliminate *any* counterexample, this set is never empty, see Theorem 23. To compute \mathcal{L} , we consider all terms that are *relevant* for the formula φ .

³ We tested several encodings, but surprisingly, this non-linear encoding worked best.

⁴ Note that we have $\llbracket \exp(0, 0) \rrbracket^{\text{EIA}} = 0^0 = 1$.

Definition 6 (Relevant Terms). A term $\text{exp}(s, t)$ is relevant if φ has a subterm of the form $\text{exp}(\pm s, \pm t)$.

Example 7 (Relevant Terms). For our leading example (2), the relevant terms are all terms of the form $\text{exp}(\pm x, \pm y^2)$, $\text{exp}(\pm y, \pm y)$, or $\text{exp}(\pm x, \pm \text{exp}(y, y))$.

While the formula φ is changed in Line 11 of Algorithm 1, we only conjoin new lemmas to φ , and thus, relevant terms can never become irrelevant. Moreover, by construction our lemmas only contain exp -terms that were already relevant before. Thus, the set of relevant terms is not changed by our CEGAR loop.

As mentioned in Sect. 1, our approach may also compute lemmas with non-linear polynomial arithmetic. However, our lemmas are linear if s is an integer constant and t is linear for all subterms $\text{exp}(s, t)$ of φ . Here, despite the fact that the function “ mod ” is not contained in the signature of LIA, we also consider literals of the form $s \text{ mod } c = 0$ where $c \in \mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ as linear. The reason is that, according to the SMT-LIB standard, LIA contains a function⁵ “ divisible_c **Int Bool**” for each $c \in \mathbb{N}_+$, which yields **true** iff its argument is divisible by c , and hence we have $s \text{ mod } c = 0$ iff $\text{divisible}_c(s)$.

In the sequel, $\llbracket \dots \rrbracket$ means $\llbracket \dots \rrbracket^{\mathbf{A}}$, where \mathbf{A} is the model from Line 6 of Algorithm 1.

4.2.1 Symmetry Lemmas *Symmetry lemmas* encode the relation between terms of the form $\text{exp}(\pm s, \pm t)$. For each relevant term $\text{exp}(s, t)$, the set \mathcal{L} contains the following symmetry lemmas:

$$t \text{ mod } 2 = 0 \implies \text{exp}(s, t) = \text{exp}(-s, t) \quad (\text{SYM}_1)$$

$$t \text{ mod } 2 = 1 \implies \text{exp}(s, t) = -\text{exp}(-s, t) \quad (\text{SYM}_2)$$

$$\text{exp}(s, t) = \text{exp}(s, -t) \quad (\text{SYM}_3)$$

Note that SYM_1 and SYM_2 are just implications, not equivalences, as, for example, $c^{|d|} = (-c)^{|d|}$ does not imply $d \text{ mod } 2 = 0$ if $c = 0$.

Example 8 (Symmetry Lemmas). For our leading example (2), the following symmetry lemmas would be considered, among others:

$$\text{SYM}_1 : \quad -y \text{ mod } 2 = 0 \implies \text{exp}(-y, -y) = \text{exp}(y, -y) \quad (3)$$

$$\text{SYM}_2 : \quad -y \text{ mod } 2 = 1 \implies \text{exp}(-y, -y) = -\text{exp}(y, -y) \quad (4)$$

$$\text{SYM}_3 : \quad \text{exp}(x, \text{exp}(y, y)) = \text{exp}(x, -\text{exp}(y, y)) \quad (5)$$

$$\text{SYM}_3 : \quad \text{exp}(y, y) = \text{exp}(y, -y) \quad (6)$$

Note that, e.g., (3) results from the term $\text{exp}(-y, -y)$, which is relevant (see Definition 6) even though it does not occur in φ .

⁵ We excluded these functions from Σ_{Int} , as they can be simulated with mod .

To show soundness of our refinement, we have to show that our lemmas are EIA-valid.

Lemma 9. *Let s, t be terms of sort **Int**. Then SYM_1 – SYM_3 are EIA-valid.*

4.2.2 Monotonicity Lemmas *Monotonicity lemmas* are of the form

$$s_2 \geq s_1 > 1 \wedge t_2 \geq t_1 > 0 \wedge (s_2 > s_1 \vee t_2 > t_1) \implies \exp(s_2, t_2) > \exp(s_1, t_1), \quad (\text{mon})$$

i.e., they prohibit violations of monotonicity of \exp .

Example 10 (Monotonicity Lemmas). For our leading example (2), we obtain, e.g., the following lemmas:

$$x > 1 \wedge \exp(y, y) > y^2 > 0 \implies \exp(x, \exp(y, y)) > \exp(x, y^2) \quad (7)$$

$$x > 1 \wedge -\exp(y, y) > y^2 > 0 \implies \exp(x, -\exp(y, y)) > \exp(x, y^2) \quad (8)$$

So for each pair of two different relevant terms $\exp(s_1, t_1), \exp(s_2, t_2)$ where $\llbracket s_2 \rrbracket \geq \llbracket s_1 \rrbracket > 1$ and $\llbracket t_2 \rrbracket \geq \llbracket t_1 \rrbracket > 0$, the set \mathcal{L} contains **mon**.

Lemma 11. *Let s_1, s_2, t_1, t_2 be terms of sort **Int**. Then **mon** is EIA-valid.*

4.2.3 Bounding Lemmas *Bounding lemmas* provide bounds on relevant terms $\exp(s, t)$ where $\llbracket s \rrbracket$ and $\llbracket t \rrbracket$ are non-negative. Together with symmetry lemmas, they also give rise to bounds for the cases where s or t are negative.

For each relevant term $\exp(s, t)$ where $\llbracket s \rrbracket$ and $\llbracket t \rrbracket$ are non-negative, the following lemmas are contained in \mathcal{L} :

$$t = 0 \implies \exp(s, t) = 1 \quad (\text{BND}_1)$$

$$t = 1 \implies \exp(s, t) = s \quad (\text{BND}_2)$$

$$s = 0 \wedge t \neq 0 \iff \exp(s, t) = 0 \quad (\text{BND}_3)$$

$$s = 1 \implies \exp(s, t) = 1 \quad (\text{BND}_4)$$

$$s + t > 4 \wedge s > 1 \wedge t > 1 \implies \exp(s, t) > s \cdot t + 1 \quad (\text{BND}_5)$$

The cases $t \in \{0, 1\}$ are also addressed by our first rewrite rule (see Sect. 4.1). However, this rewrite rule only applies if t is an integer constant. In contrast, the first two lemmas above apply if t evaluates to 0 or 1 in the current model.

Example 12 (Bounding Lemmas). For our leading example (2), the following bounding lemmas would be considered, among others:

$$\text{BND}_1 : \quad \exp(y, y) = 0 \implies \exp(x, \exp(y, y)) = 1$$

$$\text{BND}_2 : \quad \exp(y, y) = 1 \implies \exp(x, \exp(y, y)) = x$$

$$\text{BND}_3 : \quad x = 0 \wedge \exp(y, y) \neq 0 \iff \exp(x, \exp(y, y)) = 0$$

$$\text{BND}_4 : \quad x = 1 \implies \exp(x, \exp(y, y)) = 1$$

$$\text{BND}_5 : \quad y > 2 \implies \exp(y, y) > y^2 + 1 \quad (9)$$

$$\text{BND}_5 : \quad -y > 2 \implies \exp(-y, -y) > y^2 + 1 \quad (10)$$

Lemma 13. *Let s, t be terms of sort **Int**. Then $\text{BND}_1\text{--BND}_5$ are EIA-valid.*

The bounding lemmas are defined in such a way that they provide lower bounds for $\text{exp}(s, t)$ for almost all non-negative values of s and t . The reason why we focus on lower bounds is that polynomials can only bound $\text{exp}(s, t)$ from above for finitely many values of s and t . The missing (lower and upper) bounds are provided by *interpolation lemmas*.

4.2.4 Interpolation Lemmas In addition to bounding lemmas, we use *interpolation lemmas* that are constructed via *bilinear interpolation* to provide bounds. Here, we assume that the arguments of exp are positive, as negative arguments are handled by symmetry lemmas, and bounding lemmas yield tight bounds if at least one argument of exp is 0. The correctness of interpolation lemmas relies on the following observation.

Lemma 14. *Let $f : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ be convex, $w_1, w_2 \in \mathbb{R}_+$, and $w_1 < w_2$. Then*

$$\begin{aligned} \forall x \in [w_1, w_2]. f(x) &\leq f(w_1) + \frac{f(w_2) - f(w_1)}{w_2 - w_1} \cdot (x - w_1) && \text{and} \\ \forall x \in \mathbb{R}_+ \setminus (w_1, w_2). f(x) &\geq f(w_1) + \frac{f(w_2) - f(w_1)}{w_2 - w_1} \cdot (x - w_1). \end{aligned}$$

Here, $[w_1, w_2]$ and (w_1, w_2) denote closed and open real intervals. Note that the right-hand side of the inequations above is the linear interpolant of f between w_1 and w_2 . Intuitively, it corresponds to the secant of f between the points $(w_1, f(w_1))$ and $(w_2, f(w_2))$, and thus the lemma follows from convexity of f .

Let $\text{exp}(s, t)$ be relevant, $\llbracket s \rrbracket = c > 0$, $\llbracket t \rrbracket = d > 0$, and $\llbracket \text{exp} \rrbracket(c, d) \neq c^d$, i.e., we want to prohibit the current interpretation of $\text{exp}(s, t)$.

Interpolation Lemmas for Upper Bounds. First assume $\llbracket \text{exp} \rrbracket(c, d) > c^d$, i.e., to rule out this counterexample, we need a lemma that provides a suitable upper bound for $\text{exp}(c, d)$. Let $c', d' \in \mathbb{N}_+$ and:

$$\begin{aligned} c^- &:= \min(c, c') & c^+ &:= \max(c, c') & d^- &:= \min(d, d') & d^+ &:= \max(d, d') \\ [c^\pm] &:= [c^- .. c^+] & [d^\pm] &:= [d^- .. d^+] \end{aligned}$$

Here, $[a .. b]$ denotes a closed integer interval. Then we first use d^-, d^+ for linear interpolation w.r.t. the 2^{nd} argument of $\lambda x, y. x^y$. To this end, let

$$\text{ip}_2^{[d^\pm]}(x, y) := x^{d^-} + \frac{x^{d^+} - x^{d^-}}{d^+ - d^-} \cdot (y - d^-),$$

where we define $\frac{a}{b} := \frac{a}{b}$ if $b \neq 0$ and $\frac{a}{0} := 0$. So if $d^- < d^+$, then $\text{ip}_2^{[d^\pm]}(x, y)$ corresponds to the linear interpolant of x^y w.r.t. y between d^- and d^+ . Then $\text{ip}_2^{[d^\pm]}(x, y)$ is a suitable upper bound, as

$$\forall x \in \mathbb{N}_+, y \in [d^\pm]. x^y \leq \text{ip}_2^{[d^\pm]}(x, y) \tag{11}$$

follows from Lemma 14. Hence, we could derive the following EIA-valid lemma:⁶

$$s > 0 \wedge t \in [d^\pm] \implies \exp(s, t) \leq \text{ip}_2^{[d^\pm]}(s, t) \tag{IP_1}$$

Example 15 (Linear Interpolation w.r.t. y). Let $\llbracket \exp(s, t) \rrbracket = \llbracket \exp \rrbracket(3, 9) > 3^9$, i.e., we have $c = 3$ and $d = 9$. Moreover, assume $c' = d' = 1$, i.e., we get $c^- = 1$, $c^+ = 3$, $d^- = 1$, and $d^+ = 9$. Then

$$\text{ip}_2^{[d^\pm]}(x, y) = \text{ip}_2^{[1..9]}(x, y) = x^1 + \frac{x^9 - x^1}{9 - 1} \cdot (y - 1) = x + \frac{x^9 - x}{8} \cdot (y - 1).$$

Hence, IP₁ corresponds to

$$s > 0 \wedge t \in [1, 9] \implies \exp(s, t) \leq s + \frac{s^9 - s}{8} \cdot (t - 1).$$

This lemma would be violated by our counterexample, as we have

$$\left\lceil s + \frac{s^9 - s}{8} \cdot (t - 1) \right\rceil = 3 + \frac{3^9 - 3}{8} \cdot 8 = 3^9 < \llbracket \exp \rrbracket(3, 9) = \llbracket \exp(s, t) \rrbracket.$$

However, the degree of $\text{ip}_2^{[d^\pm]}(s, t)$ depends on d^+ , which in turn depends on the model that was found by the underlying SMT solver. Thus, the degree of $\text{ip}_2^{[d^\pm]}(s, t)$ can get very large, which is challenging for the underlying solver.

So we next use c^-, c^+ for linear interpolation w.r.t. the 1st argument of $\lambda x, y. x^y$, resulting in

$$\text{ip}_1^{[c^\pm]}(x, y) := (c^-)^y + \frac{(c^+)^y - (c^-)^y}{c^+ - c^-} \cdot (x - c^-).$$

Then due to Lemma 14, $\text{ip}_1^{[c^\pm]}(x, y)$ is also an upper bound on the exponentiation function, i.e., we have

$$\forall y \in \mathbb{N}_+, x \in [c^\pm]. x^y \leq \text{ip}_1^{[c^\pm]}(x, y). \tag{12}$$

Note that we have $\frac{y-d^-}{d^+-d^-} \in [0, 1]$ for all $y \in [d^\pm]$, and thus

$$\text{ip}_2^{[d^\pm]}(x, y) = x^{d^-} \cdot \left(1 - \frac{y - d^-}{d^+ - d^-} \right) + x^{d^+} \cdot \frac{y - d^-}{d^+ - d^-}$$

⁶ Strictly speaking, this lemma is not a $\Sigma_{\text{Int}}^{\text{exp}}$ -term if $d^+ > d^-$, as the right-hand side makes use of division in this case. However, an equivalent $\Sigma_{\text{Int}}^{\text{exp}}$ -term can clearly be obtained by multiplying with the divisor.

is monotonically increasing in both x^{d^-} and x^{d^+} . Hence, in the definition of $\text{ip}_2^{[d^\pm]}$, we can approximate x^{d^-} and x^{d^+} with their upper bounds $\text{ip}_1^{[c^\pm]}(x, d^-)$ and $\text{ip}_1^{[c^\pm]}(x, d^+)$ that can be derived from (12). Then (11) yields

$$\forall x \in [c^\pm], y \in [d^\pm]. x^y \leq \text{ip}^{[c^\pm][d^\pm]}(x, y) \tag{13}$$

where

$$\text{ip}^{[c^\pm][d^\pm]}(x, y) := \text{ip}_1^{[c^\pm]}(x, d^-) + \frac{\text{ip}_1^{[c^\pm]}(x, d^+) - \text{ip}_1^{[c^\pm]}(x, d^-)}{d^+ - d^-} \cdot (y - d^-).$$

So the set \mathcal{L} contains the lemma

$$s \in [c^\pm] \wedge t \in [d^\pm] \implies \text{exp}(s, t) \leq \text{ip}^{[c^\pm][d^\pm]}(s, t), \tag{IP_2}$$

which is valid due to (13), and rules out any counterexample with $\llbracket \text{exp} \rrbracket(c, d) > c^d$, as $\text{ip}^{[c^\pm][d^\pm]}(c, d) = c^d$.

Example 16 (Bilinear Interpolation, Example 15 continued). In our example, we have:

$$\begin{aligned} \text{ip}_1^{[c^\pm]}(x, y) &= \text{ip}_1^{[1..3]}(x, y) = 1^y + \frac{3^y - 1^y}{3 - 1} \cdot (x - 1) = 1 + \frac{3^y - 1}{2} \cdot (x - 1) \\ \text{ip}_1^{[c^\pm]}(s, d^-) &= \text{ip}_1^{[1..3]}(s, 1) = 1 + \frac{3 - 1}{2} \cdot (s - 1) = s \\ \text{ip}_1^{[c^\pm]}(s, d^+) &= \text{ip}_1^{[1..3]}(s, 9) = 1 + \frac{3^9 - 1}{2} \cdot (s - 1) = 1 + 9841 \cdot (s - 1) \end{aligned}$$

Hence, we obtain the lemma

$$s \in [1, 3] \wedge t \in [1, 9] \implies \text{exp}(s, t) \leq s + \frac{1 + 9841 \cdot (s - 1) - s}{8} \cdot (t - 1).$$

This lemma is violated by our counterexample, as we have

$$\left\lceil s + \frac{1 + 9841 \cdot (s - 1) - s}{8} \cdot (t - 1) \right\rceil = 3^9 < \llbracket \text{exp} \rrbracket(3, 9) = \llbracket \text{exp}(s, t) \rrbracket.$$

IP_2 relates $\text{exp}(s, t)$ with the *bilinear* function $\text{ip}^{[c^\pm][d^\pm]}(s, t)$, i.e., this function is linear w.r.t. both s and t , but it multiplies s and t . Thus, if s is an integer constant and t is linear, then the resulting lemma is linear, too.

To compute interpolation lemmas, a second point (c', d') is needed. In our implementation, we store all points (c, d) where interpolation has previously been applied and use the one which is closest to the current one. The same heuristic is used to compute *secant lemmas* in [11]. For the 1st interpolation step, we use $(c', d') = (c, d)$. In this case, IP_2 simplifies to $s = c \wedge t = d \implies \text{exp}(s, t) \leq c^d$.

Lemma 17. *Let $c^+ \geq c^- > 0$ and $d^+ \geq d^- > 0$. Then IP_2 is EIA-valid.*

Interpolation Lemmas for Lower Bounds. While bounding lemmas already yield lower bounds, the bounds provided by BND_5 are not exact, in general. Hence, if $\llbracket \text{exp} \rrbracket(c, d) < c^d$, then we also use bilinear interpolation to obtain a precise lower bound for $\text{exp}(c, d)$. Dually to (11) and (12), Lemma 14 implies:

$$\forall x, y \in \mathbb{N}_+. x^y \geq \text{ip}_2^{[d..d+1]}(x, y) \quad (14) \quad \forall x, y \in \mathbb{N}_+. x^y \geq \text{ip}_1^{[c..c+1]}(x, y) \quad (15)$$

Additionally, we also obtain

$$\forall x, y \in \mathbb{N}_+. x^{y+1} - x^y \geq \text{ip}_1^{[c..c+1]}(x, y + 1) - \text{ip}_1^{[c..c+1]}(x, y) \quad (16)$$

from Lemma 14. The reason is that for $f(x) := x^{y+1} - x^y$, the right-hand side of (16) is equal to the linear interpolant of f between c and $c + 1$. Moreover, f is convex, as $f(x) = x^y \cdot (x - 1)$ where for any fixed $y \in \mathbb{N}_+$, both x^y and $x - 1$ are non-negative, monotonically increasing, and convex on \mathbb{R}_+ .

If $y \geq d$, then $\text{ip}_2^{[d..d+1]}(x, y) = x^d + (x^{d+1} - x^d) \cdot (y - d)$ is monotonically increasing in the first occurrence of x^d , and in $x^{d+1} - x^d$. Thus, by approximating x^d and $x^{d+1} - x^d$ with their lower bounds from (15) and (16), (14) yields

$$\begin{aligned} \forall x \in \mathbb{N}_+, y \geq d. x^y &\geq \text{ip}_1^{[c..c+1]}(x, d) + (\text{ip}_1^{[c..c+1]}(x, d + 1) - \text{ip}_1^{[c..c+1]}(x, d)) \cdot (y - d) \\ &= \text{ip}^{[c..c+1][d..d+1]}(x, y). \end{aligned} \quad (17)$$

So dually to IP_2 , the set \mathcal{L} contains the lemma

$$s \geq 1 \wedge t \geq d \implies \text{exp}(s, t) \geq \text{ip}^{[c..c+1][d..d+1]}(s, t) \quad (\text{IP}_3)$$

which is valid due to (17) and rules out any counterexample with $\llbracket \text{exp} \rrbracket(c, d) < c^d$, as $\text{ip}^{[c..c+1][d..d+1]}(c, d) = c^d$.

Example 18 (Interpolation, Lower Bounds). Let $\llbracket \text{exp}(s, t) \rrbracket = \llbracket \text{exp} \rrbracket(3, 9) < 3^9$, i.e., we have $c = 3$, and $d = 9$. Then

$$\begin{aligned} \text{ip}_1^{[3..4]}(x, 9) &= 3^9 + (4^9 - 3^9) \cdot (x - 3) = 19683 + 242461 \cdot (x - 3) \\ \text{ip}_1^{[3..4]}(x, 10) &= 3^{10} + (4^{10} - 3^{10}) \cdot (x - 3) = 59049 + 989527 \cdot (x - 3) \\ \text{ip}^{[3..4][9..10]}(x, y) &= \text{ip}_1^{[3..4]}(x, 9) + (\text{ip}_1^{[3..4]}(x, 10) - \text{ip}_1^{[3..4]}(x, 9)) \cdot (y - 9) \end{aligned}$$

and thus we obtain the lemma

$$s \geq 1 \wedge t \geq 9 \implies \text{exp}(s, t) \geq 747066 \cdot s \cdot t - 6481133 \cdot s - 2201832 \cdot t + 19108788.$$

It is violated by our counterexample, as we have

$$\llbracket 747066 \cdot s \cdot t - 6481133 \cdot s - 2201832 \cdot t + 19108788 \rrbracket = 3^9 > \llbracket \text{exp} \rrbracket(3, 9).$$

Lemma 19. *Let $c, d \in \mathbb{N}_+$. Then IP_3 is EIA-valid.*

4.2.5 Lazy Lemma Generation In practice, it is not necessary to compute the entire set of lemmas \mathcal{L} . Instead, we can stop as soon as \mathcal{L} contains a single lemma which is violated by the current counterexample. However, such a strategy would result in a quite fragile implementation, as its behavior would heavily depend on the order in which lemmas are computed, which in turn depends on low-level details like the order of iteration over sets, etc. So instead, we improve Lines 9–11 of Algorithm 1 and use the following precedence on our four kinds of lemmas:

$$\text{symmetry} \succ \text{monotonicity} \succ \text{bounding} \succ \text{interpolation}$$

Then we compute all lemmas of the same kind, starting with symmetry lemmas, and we only proceed with the next kind if none of the lemmas computed so far is violated by the current counterexample. The motivation for the order above is as follows: Symmetry lemmas obtain the highest precedence, as other kinds of lemmas depend on them for restricting $\text{exp}(s, t)$ in the case that s or t is negative. As the coefficients in interpolation lemmas for $\text{exp}(s, t)$ grow exponentially w.r.t. $\llbracket t \rrbracket$ (see, e.g., Example 18), interpolation lemmas get the lowest precedence. Finally, we prefer monotonicity lemmas over bounding lemmas, as monotonicity lemmas are linear (if the arguments of exp are linear), whereas BND_5 may be non-linear.

Example 20 (Leading Example Finished). We now finish our leading example which, after preprocessing, looks as follows (see Example 4):

$$x^2 > 4 \wedge y^2 > 4 \wedge \text{exp}(x, y^2) = \text{exp}(x, \text{exp}(y, y)) \tag{2}$$

Then our implementation generates 12 symmetry lemmas, 4 monotonicity lemmas, and 8 bounding lemmas before proving unsatisfiability, including

$$(3), (4), (5), (6), (7), (8), (9), \text{ and } (10).$$

These lemmas suffice to prove unsatisfiability for the case $x > 2$ (the cases $x \in [-2..2]$ or $y \in [-2..2]$ are trivial). For example, if $y < -2$ and $-y \bmod 2 = 0$, we get

$$\begin{aligned} y < -2 &\stackrel{(10)}{\curvearrowright} \text{exp}(-y, -y) > y^2 + 1 \stackrel{(3)}{\curvearrowright} \text{exp}(y, -y) > y^2 + 1 \\ &\stackrel{(6)}{\curvearrowright} \text{exp}(y, y) > y^2 + 1 \stackrel{(7)}{\curvearrowright} \text{exp}(x, \text{exp}(y, y)) > \text{exp}(x, y^2) \stackrel{(2)}{\curvearrowright} \text{false} \end{aligned}$$

and for the cases $y > 2$ and $y < -2 \wedge -y \bmod 2 = 1$, unsatisfiability can be shown similarly. For the case $x < -2$, 5 more symmetry lemmas, 2 more monotonicity lemmas, and 3 more bounding lemmas are used. The remaining 3 symmetry lemmas and 3 bounding lemmas are not used in the final proof of unsatisfiability.

While our leading example can be solved without interpolation lemmas, in general, interpolation lemmas are a crucial ingredient of our approach.

Example 21. Consider the formula

$$1 < x < y \wedge 0 < z \wedge \exp(x, z) < \exp(y, z).$$

Our implementation first rules out 33 counterexamples using 7 bounding lemmas and 42 interpolation lemmas in ~ 0.1 seconds, before finding the model $\llbracket x \rrbracket = 21$, $\llbracket y \rrbracket = 721$, and $\llbracket z \rrbracket = 4$. Recall that interpolation lemmas are only used if a counterexample cannot be ruled out by any other kinds of lemmas. So without interpolation lemmas, our implementation could not solve this example.

Our main soundness theorem follows from soundness of our preprocessings (Lemma 5) and the fact that all of our lemmas are EIA-valid (Lemmas 9, 11, 13, 17, and 19).

Theorem 22 (Soundness of Algorithm 1). *If Algorithm 1 returns **sat**, then φ is satisfiable in EIA. If Algorithm 1 returns **unsat**, then φ is unsatisfiable in EIA.*

Another important property of Algorithm 1 is that it can eliminate *any* counterexample, and hence it makes progress in every iteration.

Theorem 23 (Progress Theorem). *If \mathbf{A} is a counterexample and \mathcal{L} is computed as in Algorithm 1, then*

$$\mathbf{A} \not\models \bigwedge \mathcal{L}.$$

Despite Theorems 22 and 23, EIA is of course undecidable, and hence Algorithm 1 is incomplete. For example, it does not terminate for the input formula

$$y \neq 0 \wedge \exp(2, x) = \exp(3, y). \quad (18)$$

Here, to prove unsatisfiability, one needs to know that $2^{|x|}$ is 1 or even, but $3^{|y|}$ is odd and greater than 1 (unless $y = 0$). This cannot be derived from the lemmas used by our approach. Thus, Algorithm 1 would refine the formula (18) infinitely often.

Note that monotonicity lemmas are important, even though they are not required to prove Theorem 23. The reason is that *all* (usually infinitely many) counterexamples must be eliminated to prove **unsat**. For instance, reconsider Example 20, where the monotonicity lemma (7) eliminates infinitely many counterexamples with $\llbracket \exp(x, \exp(y, y)) \rrbracket \leq \llbracket \exp(x, y^2) \rrbracket$. In contrast, Theorem 23 only guarantees that every single counterexample can be eliminated. Consequently, our implementation does not terminate on our leading example if monotonicity lemmas are disabled.

5 Related Work

The most closely related work applies *incremental linearization* to NIA, or to non-linear real arithmetic with transcendental functions (NRAT). Like our approach, incremental linearization is an instance of the CEGAR paradigm: An initial abstraction (where certain predefined functions are considered as uninterpreted functions) is refined via linear lemmas that rule out the current counterexample.

Our approach is inspired by, but differs significantly from the approach for linearization of NRAT from [11]. There, non-linear polynomials are linearized as well, whereas we leave the handling of polynomials to the backend solver. Moreover, [11] uses linear lemmas only, whereas we also use bilinear lemmas. Furthermore, [11] fixes the base to Euler's number e , whereas we consider a binary version of exponentiation.

The only lemmas that easily carry over from [11] are monotonicity lemmas. While [11] also uses symmetry lemmas, they express properties of the sine function, i.e., they are fundamentally different from ours. Our bounding lemmas are related to the “lower bound” and “zero” lemmas from [11], but there, $\lambda x. e^x$ is trivially bounded by 0. Interpolation lemmas are related to the “tangent” and “secant lemmas” from [11]. However, tangent lemmas make use of first derivatives, so they are not expressible with integer arithmetic in our setting, as we have $\frac{\partial}{\partial y} x^y = x^y \cdot \ln x$. Secant lemmas are essentially obtained by linear interpolation, so our interpolation lemmas can be seen as a generalization of secant lemmas to binary functions. A preprocessing by rewriting is not considered in [11].

In [10], incremental linearization is applied to NIA. The lemmas that are used in [10] are similar to those from [11], so they differ fundamentally from ours, too.

Further existing approaches for NRAT are based on interval propagation [14, 24]. As observed in [11], interval propagation effectively computes a piecewise *constant* approximation, which is less expressive than our bilinear approximations.

Recently, a novel approach for NRAT based on the *topological degree test* has been proposed [12, 30]. Its strength is finding irrational solutions more often than other approaches for NRAT. Hence, this line of work is orthogonal to ours.

EIA could also be tackled by combining NRAT techniques with branch-and-bound, but the following example shows that doing so is not promising.

Example 24. Consider the formula $x = \exp(3, y) \wedge y > 0$. To tackle it with existing solvers, we have to encode it using the natural exponential function:

$$e^z = 3 \wedge x = e^{y \cdot z} \wedge y > 0 \tag{19}$$

Here x and y range over the integers and z ranges over the reals. Any model of (19) satisfies $z = \ln 3$, where $\ln 3$ is irrational. As finding such models is challenging, the leading tools MathSat [9] and CVC5 [2] fail for $e^z = 3$.

MetiTarski [1] integrates decision procedures for real closed fields and approximations for transcendental functions into the theorem prover Metis [27] to prove theorems about the reals. In a related line of work, iSAT3 [14] has been coupled with SPASS [35]. Clearly, these approaches differ fundamentally from ours.

Recently, the complexity of a decidable extension of linear integer arithmetic with exponentiation has been investigated [5]. It is equivalent to EIA without the functions “.”, “div”, and “mod”, and where the first argument of all occurrences of exp must be the same constant. Integrating decision procedures for fragments like this one into our approach is an interesting direction for future work.

6 Implementation and Evaluation

Implementation. We implemented our approach in our novel tool SwlnE. It is based on SMT-Switch [31], a library that offers a unified interface for various SMT solvers. SwlnE uses the backend solvers Z3 4.12.2 [32] and CVC5 1.0.8 [2]. It supports incrementality and can compute models for variables, but not yet for uninterpreted functions, due to limitations inherited from SMT-Switch.

The backend solver (which defaults to Z3) can be selected via command-line flags. For more information on SwlnE and a precompiled release, we refer to [22, 23].

Benchmarks. To evaluate our approach, we synthesized a large collection of EIA problems from verification benchmarks for safety, termination, and complexity analysis. More precisely, we ran our verification tool LoAT [18] on the benchmarks for *linear Constrained Horn Clauses (CHCs)* with *linear integer arithmetic* from the *CHC Competitions 2022 and 2023* [8] as well as on the benchmarks for *Termination and Complexity of Integer Transition Systems* from the *Termination Problems Database (TPDB)* [34], the benchmark set of the *Termination and Complexity Competition* [25], and extracted all SMT problems with exponentiation that LoAT created while analyzing these benchmarks. Afterwards, we removed duplicates.

The resulting benchmark set consists of 4627 SMT problems, which are available at [22]:

- 669 problems that resulted from the benchmarks of the CHC Competition ’22 (called *CHC Comp ’22 Problems* below)
- 158 problems that resulted from the benchmarks of the CHC Competition ’23 (*CHC Comp ’23 Problems*)
- 3146 problems that resulted from the complexity benchmarks of the TPDB (*Complexity Problems*)
- 654 problems that resulted from the termination benchmarks of the TPDB (*Termination Problems*)

Evaluation. We ran SwlnE with both supported backend solvers (Z3 and CVC5). To evaluate the impact of the different components of our approach, we

also tested with configurations where we disabled rewriting, symmetry lemmas, bounding lemmas, interpolation lemmas, or monotonicity lemmas. All experiments were performed on StarExec [33] with a wall clock timeout of 10s and a memory limit of 128GB per example. We chose a small timeout, as LoAT usually has to discharge many SMT problems to solve a single verification task. So in our setting, each individual SMT problem should be solved quickly.

The results can be seen in Tables 1, 2, 3, and 4, where VB means “virtual best”. All but 48 of the 4627 benchmarks can be solved, and all unsolved benchmarks are Complexity Problems. All CHC Comp Problems can be solved with both backend solvers. Considering Complexity and Termination Problems, Z3

Table 1. CHC Comp '22 – Results

backend	configuration	sat	unsat	unknown
Z3	default	296	373	0
CVC5		296	373	0
VB		296	373	0
Z3	no rewriting	291	373	5
	no symmetry	296	373	0
	no bounding	110	373	186
	no interpolation	3	372	294
	no monotonicity	296	373	0
	no rewriting, no lemmas	1	364	304
CVC5	no rewriting	296	372	1
	no symmetry	296	373	0
	no bounding	186	373	110
	no interpolation	28	372	269
	no monotonicity	296	373	0
	no rewriting, no lemmas	1	364	304

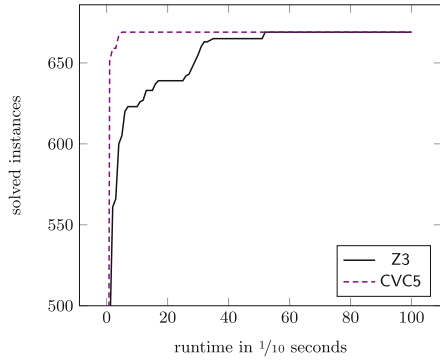


Fig. 1. CHC Comp '22 – Runtime

Table 2. CHC Comp '23 – Results

backend	configuration	sat	unsat	unknown
Z3	default	87	71	0
CVC5		87	71	0
VB		87	71	0
Z3	no rewriting	86	71	1
	no symmetry	87	71	0
	no bounding	79	71	8
	no interpolation	0	71	87
	no monotonicity	87	71	0
	no rewriting, no lemmas	0	61	97
CVC5	no rewriting	87	71	0
	no symmetry	87	71	0
	no bounding	79	71	8
	no interpolation	36	71	51
	no monotonicity	87	71	0
	no rewriting, no lemmas	0	61	97

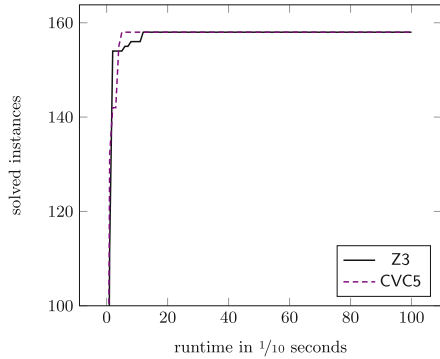


Fig. 2. CHC Comp '23 – Runtime

Table 3. Complexity – Results

backend	configuration	sat	unsat	unknown
Z3	default	1282	1789	75
CVC5		990	1784	372
VB		1309	1789	48
Z3	no rewriting	1201	1789	156
	no symmetry	975	1789	382
	no bounding	674	1788	684
	no interpolation	586	1787	773
	no monotonicity	1284	1789	73
	no rewriting, no lemmas	30	1733	1383
CVC5	no rewriting	900	1784	462
	no symmetry	954	1784	408
	no bounding	181	1784	1181
	no interpolation	405	1782	959
	no monotonicity	795	1784	567
	no rewriting, no lemmas	30	1728	1388

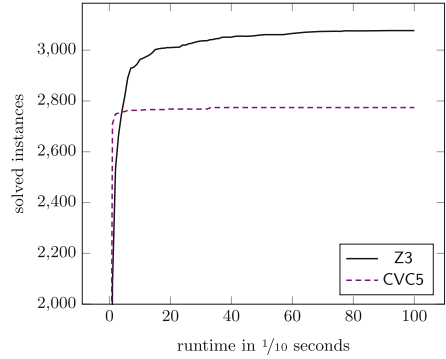


Fig. 3. Complexity – Runtime

Table 4. Termination – Results

backend	configuration	sat	unsat	unknown
Z3	default	223	431	0
CVC5		208	430	16
VB		223	431	0
Z3	no rewriting	223	431	0
	no symmetry	223	431	0
	no bounding	177	429	48
	no interpolation	15	429	210
	no monotonicity	223	431	0
	no rewriting, no lemmas	7	428	219
CVC5	no rewriting	208	430	16
	no symmetry	208	430	16
	no bounding	171	428	55
	no interpolation	10	428	216
	no monotonicity	208	430	16
	no rewriting, no lemmas	7	428	219

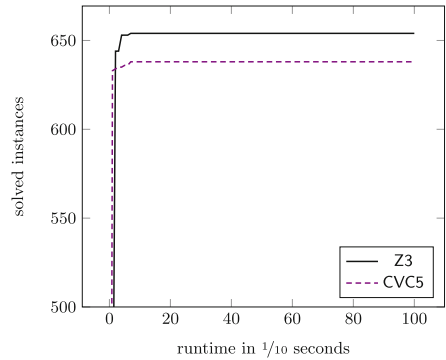


Fig. 4. Termination – Runtime

and CVC5 perform almost equally well on unsatisfiable instances, but Z3 solves more satisfiable instances.

Regarding the different components of our approach, our evaluation shows that the impact of rewriting is quite significant. For example, it enables Z3 to solve 81 additional Complexity Problems. Symmetry lemmas enable Z3 to solve more Complexity Problems, but they are less helpful for CVC5. In fact, symmetry lemmas are needed for most of the examples where Z3 succeeds but CVC5 fails, so they seem to be challenging for CVC5, presumably due to the use of “mod”. Bounding and interpolation lemmas are crucial for proving satisfiability. In particular, disabling interpolation lemmas harms more than disabling any

other feature, which shows their importance. For example, Z3 can only prove satisfiability of 3 CHC Comp Problems without interpolation lemmas.

Interestingly, only CVC5 benefits from monotonicity lemmas, which enable it to solve more Complexity Problems. From our experience, CVC5 explores the search space in a more systematic way than Z3, so that subsequent candidate models often have a similar structure. Then monotonicity lemmas can help CVC5 to find structurally different candidate models.

Remarkably, disabling a single component does not reduce the number of **unsat**'s significantly. Thus, we also evaluated configurations where *all* components were disabled, so that **exp** is just an uninterpreted function. This reduces the number of **sat** results dramatically, but most **unsat** instances can still be solved. Hence, most of them do not require reasoning about exponentials, so it would be interesting to obtain instances where proving **unsat** is more challenging.

The runtime of SwlnE can be seen in Figs. 1, 2, 3, and 4. Most instances can be solved in a fraction of a second, as desired for our use case. Moreover, CVC5 can solve more instances in the first half second, but Z3 can solve more instances later on. We refer to [22] for more details on our evaluation.

Validation. We implemented sanity checks for both **sat** and **unsat** results. For **sat**, we evaluate the input problem using EIA semantics for **exp**, and the current model for all variables. For **unsat**, assume that the input problem φ contains the subterms $\mathbf{exp}(s_0, t_0), \dots, \mathbf{exp}(s_n, t_n)$. Then we enumerate all SMT problems

$$\varphi \wedge \bigwedge_{i=0}^n t_i = c_i \wedge \mathbf{exp}(s_i, t_i) = s_i^{c_i} \quad \text{where } c_1, \dots, c_n \in [0..k] \text{ for some } k \in \mathbb{N}$$

(we used $k = 10$). If any of them is satisfiable in NIA, then φ is satisfiable in EIA. None of these checks revealed any problems.

7 Conclusion

We presented the novel SMT theory EIA, which extends the theory *non-linear integer arithmetic* with integer exponentiation. Moreover, inspired by *incremental linearization* for similar extensions of *non-linear real arithmetic*, we developed a CEGAR approach to solve EIA problems. The core idea of our approach is to regard exponentiation as an uninterpreted function and to eliminate counterexamples, i.e., models that violate the semantics of exponentiation, by generating suitable *lemmas*. Here, the use of *bilinear interpolation* turned out to be crucial, both in practice (see our evaluation in Sect. 6) and in theory, as interpolation lemmas are essential for being able to eliminate *any* counterexample (see Theorem 23). Finally, we evaluated the implementation of our approach in our novel tool SwlnE on thousands of EIA problems that were synthesized from verification tasks using our verification tool LoAT. Our evaluation shows that SwlnE is highly effective for our use case, i.e., as backend for LoAT. Hence, we will couple SwlnE and LoAT in future work.

With `SwInE`, we provide an SMT-LIB compliant open-source solver for EIA [23]. In this way, we hope to attract users with applications that give rise to challenging benchmarks, as our evaluation suggests that our benchmarks are relatively easy to solve. Moreover, we hope that other solvers with support for integer exponentiation will follow, with the ultimate goal of standardizing EIA.

References

1. Akbarpour, B., Paulson, L.C.: `MetiTarski`: an automatic theorem prover for real-valued special functions. *J. Autom. Reason.* **44**(3), 175–205 (2010). <https://doi.org/10.1007/S10817-009-9149-2>
2. Barbosa, H., et al.: `CVC5`: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) *TACAS 2022*. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: `FAST`: acceleration from theory to practice. *Int. J. Softw. Tools Technol. Transf.* **10**(5), 401–424 (2008). <https://doi.org/10.1007/s10009-008-0064-3>
4. Barrett, C., Fontaine, P., Tinelli, C.: `The Satisfiability Modulo Theories Library (SMT-LIB)`. (2016). <https://smt-lib.org/>
5. Benedikt, M., Chistikov, D., Mansutti, A.: The complexity of Presburger arithmetic with power or powers. In: Etesami, K., Feige, U., Puppis, G. (eds.) *ICALP 2023*. LIPIcs, vol. 261, pp. 112:1–112:18 (2023). <https://doi.org/10.4230/LIPIcs.ICALP.2023.112>
6. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) *CAV 2010*. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23
7. Bozga, M., Iosif, R., Konečný, F.: Relational analysis of integer programs. Technical report, TR-2012-10, VERIMAG (2012). <https://www-verimag.imag.fr/TR/TR-2012-10.pdf>
8. CHC Competition. <https://chc-comp.github.io>
9. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: `The MathSAT5 SMT solver`. In: Piterman, N., Smolka, S.A. (eds.) *TACAS 2013*. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
10. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) *SAT 2018*. LNCS, vol. 10929, pp. 383–398. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_23
11. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Trans. Comput. Log.* **19**(3), 19:1–19:52 (2018). <https://doi.org/10.1145/3230639>
12. Cimatti, A., Griggio, A., Lipparini, E., Sebastiani, R.: Handling polynomial and transcendental functions in SMT via unconstrained optimisation and topological degree test. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) *ATVA 2022*. LNCS, vol. 13505, pp. 137–153. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19992-9_9

13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
14. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *J. Satisf. Boolean Model. Comput.* **1**(3–4), 209–236 (2007). <https://doi.org/10.3233/sat190012>
15. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Barrett, C.W., Yang, J. (eds.) FMCAD 2019, pp. 221–230 (2019). <https://doi.org/10.23919/FMCAD.2019.8894271>
16. Frohn, F., Giesl, J.: Termination of triangular integer loops is decidable. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 426–444. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_24
17. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. *ACM Trans. Program. Lang. Syst.* **42**(3), 13:1–13:50 (2020). <https://doi.org/10.1145/3410331>
18. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 712–722. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_41
19. Frohn, F., Giesl, J.: Proving non-termination by acceleration driven clause learning. In: Pientka, B., Tinelli, C. (eds.) CADE 2023. LNCS, vol. 14132, pp. 220–233. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_13
20. Frohn, F., Giesl, J.: ADCL: acceleration driven clause learning for constrained horn clauses. In: Hermenegildo, M.V., Morales, J.F. (eds.) SAS 2023. LNCS, vol. 14284, pp. 259–285. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_13
21. Frohn, F., Giesl, J.: Satisfiability modulo exponential integer arithmetic. *CoRR* abs/2402.01501 (2024). <https://doi.org/10.48550/arXiv.2402.01501>
22. Frohn, F., Giesl, J.: Evaluation of “Satisfiability modulo exponential integer arithmetic” (2024). <https://aprove-developers.github.io/swine-eval/>
23. Frohn, F., Giesl, J.: SwInE (2024). <https://ffrohn.github.io/swine/>
24. Gao, S., Avigad, J., Clarke, E.M.: δ -complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 286–300. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_23
25. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 156–166. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_10
26. Hark, M., Frohn, F., Giesl, J.: Termination of triangular polynomial loops. *Form. Methods Syst. Des.* (2023). <https://doi.org/10.1007/s10703-023-00440-z>
27. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: STRATA 2003, pp. 56–68. Report NASA/CP-2003-212448 (2003). <https://apps.dtic.mil/sti/pdfs/ADA418902.pdf>
28. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Jagannathan, S., Sewell, P. (eds.) POPL 2014, pp. 529–540 (2014). <https://doi.org/10.1145/2535838.2535843>

29. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. *Formal Methods Syst. Des.* **47**(1), 75–92 (2015). <https://doi.org/10.1007/s10703-015-0228-1>
30. Lipparini, E., Ratschan, S.: Satisfiability of non-linear transcendental arithmetic as a certificate search problem. In: Rozier, K.Y., Chaudhuri, S. (eds.) *NFM 2023*. LNCS, vol. 13903, pp. 472–488. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33170-1_29
31. Mann, M., et al.: *SMT-Switch: a solver-agnostic C++ API for SMT solving*. In: Li, C.-M., Manyà, F. (eds.) *SAT 2021*. LNCS, vol. 12831, pp. 377–386. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_26
32. de Moura, L., Bjørner, N.: *Z3: an efficient SMT solver*. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Stump, A., Sutcliffe, G., Tinelli, C.: *StarExec: a cross-community infrastructure for logic solving*. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) *IJCAR 2014*. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28
34. Termination Problems Data Base (TPDB). <http://termination-portal.org/wiki/TPDB>
35. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischniewski, P.: *SPASS version 3.5*. In: Schmidt, R.A. (ed.) *CADE 2009*. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10

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

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





SAT-Based Learning of Computation Tree Logic

Adrien Pommellet^(✉) , Daniel Stan , and Simon Scatton

LRE, EPITA, Le Kremlin-Bicêtre, France

{adrien,simon.scatton}@lrde.epita.fr, daniel.stan@epita.fr
<https://www.lrde.epita.fr/adrien/>, <https://www.tudo.re/daniel.stan/>

Abstract. The CTL learning problem consists in finding for a given sample of positive and negative Kripke structures a distinguishing CTL formula that is verified by the former but not by the latter. Further constraints may bound the size and shape of the desired formula or even ask for its minimality in terms of syntactic size. This synthesis problem is motivated by explanation generation for dissimilar models, e.g. comparing a faulty implementation with the original protocol. We devise a SAT-based encoding for a fixed size CTL formula, then provide an incremental approach that guarantees minimality. We further report on a prototype implementation whose contribution is twofold: first, it allows us to assess the efficiency of various output fragments and optimizations. Secondly, we can experimentally evaluate this tool by randomly mutating Kripke structures or syntactically introducing errors in higher-level models, then learning CTL distinguishing formulas.

Keywords: Computation Tree Logic · Passive learning · SAT solving

1 Introduction

Passive learning is the act of computing a theoretical model of a system from a given set of data, without being able to acquire further information by actively querying said system. The input data may have been gathered through monitoring, collecting executions and outputs of systems. Automata and logic formulas tend to be the most common models, as they allow one to better explain systems of complex or even entirely opaque design.

Linear-time Temporal Logic LTL [19] remains one of the most widely used formalisms for specifying temporal properties of reactive systems. It applies to finite or infinite execution traces, and for that reason fits the passive learning framework very well: a LTL formula is a concise way to distinguish between correct and incorrect executions. The LTL learning problem, however, is anything but trivial: even simple fragments on finite traces are NP-complete [10], and consequently recent algorithms tend to leverage SAT solvers [16].

Computation Tree Logic CTL [8] is another relevant formalism that applies to execution trees instead of isolated linear traces. It is well-known [1, Thm. 6.21] that LTL and CTL are incomparable: the former is solely defined on the resulting runs of a system, whereas the latter depends on its branching structure.

However, the CTL passive learning problem has seldom been studied in as much detail as LTL. In this article, we formalize it on *Kripke structures* (Ks): finite graph-like representations of programs. Our goal is to find a CTL formula (said to be *separating*) that is verified by every state in a positive set S^+ yet rejected by every state in a negative set S^- .

We first prove that an explicit formula can always be computed and we bound its size, assuming the sample is devoid of contradictions. However, said formula may not be minimal. The next step is therefore to solve the bounded learning problem: finding a separating CTL formula of size smaller than a given bound n . We reduce it to an instance Φ_n of the Boolean satisfiability problem whose answer can be computed by a SAT solver; to do so, we encode CTL's bounded semantics, as the usual semantics on infinite executions trees can lead to spurious results. Finally, we use a bottom-up approach to pinpoint the minimal answer by solving a series of satisfiability problems. We show that a variety of optimizations can be applied to this iterative algorithm. These various approaches have been implemented in a C++ tool and benchmarked on a test sample.

Related Work. Bounded model checking harnesses the efficiency of modern SAT solvers to iteratively look for a witness of bounded size that would contradict a given logic formula, knowing that there exists a completeness threshold after which we can guarantee no counter-example exists. First introduced by Biere et al. [3] for LTL formulas, it was later applied to CTL formulas [17, 24, 25].

This approach inspired Neider et al. [16], who designed a SAT-based algorithm that can learn a LTL formula consistent with a sample of *ultimately periodic* words by computing propositional Boolean formulas that encode both the semantics of LTL on the input sample and the syntax of its Directed Acyclic Graph (DAG) representation. This work spurred further SAT-based developments such as learning formulas in the property specification language PSL [21] or LTL_f [7], applying MaxSAT solving to noisy datasets [11], or constraining the shape of the formula to be learnt [15]. Our article extends this method to CTL formulas and Kripke structures. It subsumes the original LTL learning problem: one can trivially prove that it is equivalent to learning CTL formulas consistent with a sample of lasso-shaped Ks that consist of a single linear sequence of states followed by a single loop.

Fijalkow et al. [10] have studied the complexity of learning LTL_f formulas of size smaller than a given bound and consistent with a sample of *finite* words: it is already NP-complete for fragments as simple as $LTL_f(\wedge, X)$, $LTL_f(\wedge, F)$, or $LTL_f(F, X, \wedge, \vee)$. However, their proofs cannot be directly extended to samples of infinite but ultimately periodic words.

Browne et al. [6] proved that Ks could be characterized by CTL formulas and that conversely bisimilar Ks verified the same set of CTL formulas. As we

will show in Sect. 3, this result guarantees that a solution to the CTL learning problem actually exists if the input sample is consistent.

Wasylkowski et al. [23] mined CTL specifications in order to explain preconditions of Java functions beyond pure state reachability. However, their learning algorithm consists in enumerating CTL templates of the form $\forall F a, \exists F a, \forall G(a \implies \forall X \forall F b)$ and $\forall G(a \implies \exists X \exists F b)$ where $a, b \in \text{AP}$ for each function, using model checking to select one that is verified by the Kripke structure representing the aforementioned function.

Two very recent articles, yet to be published, have addressed the CTL learning problem as well. Bordais et al. [5] proved that the passive learning problem for LTL formulas on ultimately periodic words is NP-hard, assuming the size of the alphabet is given as an input; they then extend this result to CTL passive learning, using a straightforward reduction of ultimately periodic words to lasso-shaped Kripke structures. Roy et al. [22] used a SAT-based algorithm, resulting independently to our own research in an encoding similar to the one outlined in Sect. 4. However, our explicit solution to the learning problem, the embedding of the negations in the syntactic DAG, the approximation of the recurrence diameter as a semantic bound, our implementation of this algorithm, its test suite, and the experimental results are entirely novel contributions.

2 Preliminary Definitions

2.1 Kripke Structures

Let AP be a finite set of atomic propositions. A Kripke structure is a finite directed graph whose vertices (called *states*) are labelled by subsets of AP.

Definition 1 (Kripke Structure). A Kripke structure (KS) \mathcal{K} on AP is a tuple $\mathcal{K} = (Q, \delta, \lambda)$ such that:

- Q is a finite set of states; the integer $|Q|$ is known as the size of \mathcal{K} ;
- $\delta : Q \rightarrow (2^Q \setminus \{\emptyset\})$ is a transition function; the integer $\max_{q \in Q} |\delta(q)|$ is known as the degree of \mathcal{K} ;
- $\lambda : Q \rightarrow 2^{\text{AP}}$ is a labelling function.

An infinite run r of \mathcal{K} starting from a state $q \in Q$ is an infinite sequence $r = (s_i) \in Q^\omega$ of consecutive states such that $s_0 = q$ and $\forall i \geq 0, s_{i+1} \in \delta(s_i)$. $\mathcal{R}_{\mathcal{K}}(q)$ is the set of all infinite runs of \mathcal{K} starting from q .

The *recurrence diameter* $\alpha_{\mathcal{K}}(q)$ of state q in \mathcal{K} is the length of the longest finite run $(s_i)_{i=0, \dots, \alpha_{\mathcal{K}}(q)}$ starting from q such that $\forall i, j \in [0 \dots \alpha_{\mathcal{K}}(q)]$, if $i \neq j$ then $s_i \neq s_j$ (i.e. the longest *simple* path in the underlying graph structure). We may omit the index \mathcal{K} whenever contextually obvious.

Note that two states may generate the same runs despite their computation trees not corresponding. It is therefore necessary to define an equivalence relation on states of KSs that goes further than mere run equality.

Definition 2 (Bisimulation Relation). Let $\mathcal{K} = (Q, \delta, \lambda)$ be a KS on AP. The canonical bisimulation relation $\sim \subseteq Q \times Q$ is the coarsest (i.e. the most general) equivalence relation such that for any $q_1 \sim q_2$, $\lambda(q_1) = \lambda(q_2)$ and $\forall q'_1 \in \delta(q_1), \exists q'_2 \in \delta(q_2)$ such that $q'_1 \sim q'_2$.

Bisimilarity does not only entails equality of runs, but also similarity of shape: two bisimilar states have corresponding computation trees at any depth. A partition refinement algorithm allows one to compute \sim by refining a sequence of equivalence relations $(\sim_i)_{i \geq 0}$ on $Q \times Q$ inductively, where for every $q_1, q_2 \in Q$:

$$q_1 \sim_0 q_2 \iff \lambda(q_1) = \lambda(q_2)$$

$$q_1 \sim_{i+1} q_2 \iff (q_1 \sim_i q_2) \wedge (\{[q'_1]_{\sim_i} \mid q'_1 \in \delta(q_1)\} = \{[q'_2]_{\sim_i} \mid q'_2 \in \delta(q_2)\})$$

Where $[q]_{\sim_i}$ stands for the equivalence class of $q \in Q$ according to the equivalence relation \sim_i . Intuitively, $q_1 \sim_i q_2$ if their computation trees are corresponding up to depth i . The next theorem is a well-known result [1, Alg. 31]:

Theorem 1 (Characteristic Number). Given a KS \mathcal{K} , there exists $i_0 \in \mathbb{N}$ such that $\forall i \geq i_0, \sim = \sim_i$. The smallest integer i_0 verifying that property is known as the characteristic number $\mathcal{C}_{\mathcal{K}}$ of \mathcal{K} .

Note that Browne et al. [6] introduced an equivalent definition: the characteristic number of a KS is also the smallest integer $\mathcal{C}_{\mathcal{K}} \in \mathbb{N}$ such that any two states are not bisimilar if and only if their labelled computation trees of depth $\mathcal{C}_{\mathcal{K}}$ are not corresponding.

2.2 Computation Tree Logic

Definition 3 (Computation Tree Logic). Computation Tree Logic (CTL) is the set of formulas defined by the following grammar, where $a \in \text{AP}$ is any atomic proposition and $\dagger \in \{\forall, \exists\}$ a quantifier:

$$\varphi ::= a \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \dagger X\varphi \mid \dagger F\varphi \mid \dagger G\varphi \mid \dagger \varphi U \varphi$$

Given $E \subseteq \{\neg, \wedge, \vee, \forall X, \exists X, \forall F, \exists F, \forall G, \exists G, \forall U, \exists U\}$, we define the (syntactic) fragment $CTL(E)$ as the subset of CTL formulas featuring only operators in E .

CTL formulas are verified against states of KSs (a process known as *model checking*). Intuitively, \forall (*all*) means that all runs starting from state q must verify the property that follows, \exists (*exists*), that at least one run starting from q must verify the property that follows, $X\varphi$ (*next*), that the next state of the run must verify φ , $F\varphi$ (*finally*), that there exists a state of run verifying φ , $G\varphi$ (*globally*), that each state of the run must verify φ , and $\varphi U \psi$ (*until*), that the run must keep verifying φ at least until ψ is eventually verified.

More formally, for a state $q \in Q$ of a KS $\mathcal{K} = (Q, \delta, \lambda)$ and a CTL formula φ , we write $(q \models_{\mathcal{K}} \varphi)$ when \mathcal{K} satisfies φ . CTL's semantics are defined inductively on φ (see [1, Def. 6.4] for a complete definition); we recall below the *until* case:

Definition 4 (Semantics of $\forall U, \exists U$). Let $\mathcal{K} = (Q, \delta, \lambda)$ be a KS, φ and ψ two CTL formulas, $q \in Q$, and $\dagger \in \{\forall, \exists\}$. Then:

$$q \models_{\mathcal{K}} \dagger \varphi U \psi \iff \dagger(s_i) \in \mathcal{R}_{\mathcal{K}}(q), \exists i \geq 0, (s_i \models_{\mathcal{K}} \psi) \wedge (\forall j < i, s_j \models_{\mathcal{K}} \varphi)$$

Bisimilarity and CTL equivalence coincide [1, Thm. 7.20] on finite KSs. The proof relies on the following concept:

Theorem 2 (Browne et al. [6]). Given a KS $\mathcal{K} = (Q, \delta, \lambda)$ and a state $q \in Q$, there exists a CTL formula $\varphi_q \in \text{CTL}(\{\neg, \wedge, \vee, \forall X, \exists X\})$ known as the master formula of state q such that, for any $q' \in Q$, $q' \models_{\mathcal{K}} \varphi_q$ if and only if $q \sim q'$.

To each CTL formula φ , we associate a syntactic tree \mathcal{T} . For brevity's sake, we consider a syntactic *directed acyclic graph* (DAG) \mathcal{D} by coalescing identical subtrees in the original syntactic tree \mathcal{T} , as shown in Fig. 1. The size $|\varphi|$ of a CTL formula φ is then defined as the number of nodes of its smallest syntactic DAG. As an example, $|\neg a \wedge \forall X a| = 4$.

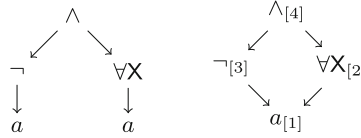


Fig. 1. The syntactic tree and indexed DAG of the CTL formula $\neg a \wedge \forall X a$.

2.3 Bounded Semantics

We introduce the bounded temporal operators $\forall F^u, \exists F^u, \forall G^u, \exists G^u, \forall U^u$, and $\exists U^u$, whose semantics only applies to the first u steps of a run. Formally:

Definition 5 (Bounded Semantics of CTL). Let $\mathcal{K} = (Q, \delta, \lambda)$ be a KS, φ and ψ two CTL formulas, $u \in \mathbb{N}$ and $q \in Q$. The bounded semantics of CTL of rank u with regards to \mathcal{K} are defined as follows for the quantifier $\dagger \in \{\forall, \exists\}$:

$$q \models_{\mathcal{K}} \dagger F^u \varphi \iff \dagger(s_i) \in \mathcal{R}_{\mathcal{K}}(q), \exists i \in [0 .. u], s_i \models_{\mathcal{K}} \varphi$$

$$q \models_{\mathcal{K}} \dagger G^u \varphi \iff \dagger(s_i) \in \mathcal{R}_{\mathcal{K}}(q), \forall i \in [0 .. u], s_i \models_{\mathcal{K}} \varphi$$

$$q \models_{\mathcal{K}} \dagger \varphi U^u \psi \iff \dagger(s_i) \in \mathcal{R}_{\mathcal{K}}(q), \exists i \in [0 .. u], (s_i \models_{\mathcal{K}} \psi) \wedge (\forall j < i, s_j \models_{\mathcal{K}} \varphi)$$

Intuitively, the rank u of the bounded semantics acts as a timer: ($q \models_{\mathcal{K}} \forall G^u \varphi$) means that φ must hold for the next u computation steps; ($q \models_{\mathcal{K}} \forall F^u \varphi$), that q must always be able to reach a state verifying φ within u steps; ($q \models_{\mathcal{K}} \forall \varphi U^u \psi$), that q must always be able to reach a state verifying ψ within u steps, and that φ must hold until it does; etc. This intuition results in the following properties:

Property 1 (Base case). $(q \models_{\mathcal{K}} \psi) \iff (q \models_{\mathcal{K}} \dagger F^0 \psi) \iff (q \models_{\mathcal{K}} \dagger \varphi U^0 \psi) \iff (q \models_{\mathcal{K}} \dagger G^0 \psi)$.

Property 2 (Induction). $(q \models_{\mathcal{K}} \dagger F^{u+1} \varphi) \iff (q \models_{\mathcal{K}} \varphi) \vee \bigtriangleup_{q' \in \delta(q)} (q' \models_{\mathcal{K}} \dagger F^u \varphi)$,

$(q \models_{\mathcal{K}} \dagger \varphi U^{u+1} \psi) \iff (q \models_{\mathcal{K}} \psi) \vee \left[(q \models_{\mathcal{K}} \varphi) \wedge \bigtriangleup_{q' \in \delta(q)} (q' \models_{\mathcal{K}} \dagger \varphi U^u \psi) \right]$, and

$(q \models_{\mathcal{K}} \dagger \mathbf{G}^{u+1} \varphi) \iff (q \models_{\mathcal{K}} \varphi) \wedge \bigtriangleup_{q' \in \delta(q)} (q' \models_{\mathcal{K}} \dagger \mathbf{G}^u \varphi)$, where $\bigtriangleup = \wedge$ if $\dagger = \forall$ and $\bigtriangleup = \vee$ if $\dagger = \exists$.

Property 3 (Spread). $(q \models_{\mathcal{K}} \dagger \mathbf{F}^u \varphi) \implies (q \models_{\mathcal{K}} \dagger \mathbf{F}^{u+1} \varphi)$, $(q \models_{\mathcal{K}} \dagger \mathbf{G}^{u+1} \varphi) \implies (q \models_{\mathcal{K}} \dagger \mathbf{G}^u \varphi)$, and $(q \models_{\mathcal{K}} \dagger \varphi \mathbf{U}^u \psi) \implies (q \models_{\mathcal{K}} \dagger \varphi \mathbf{U}^{u+1} \psi)$.

Bounded model checking algorithms [25] rely on the following result, as one can then restrict the study of CTL semantics to finite and fixed length paths.

Theorem 3. *Given $q \in Q$, for $\dagger \in \{\forall, \exists\}$ and $\triangleright \in \{\mathbf{F}, \mathbf{G}\}$, $q \models_{\mathcal{K}} \dagger \triangleright \varphi$ (resp. $q \models_{\mathcal{K}} \dagger \varphi \mathbf{U} \psi$) if and only if $q \models_{\mathcal{K}} \dagger \triangleright^{\alpha(q)} \varphi$ (resp. $q \models_{\mathcal{K}} \dagger \varphi \mathbf{U}^{\alpha(q)} \psi$).*

A full proof of this result is available in Appendix A.

3 The Learning Problem

We consider the synthesis problem of a distinguishing CTL formula from a sample of positive and negative states of a given KS.

3.1 Introducing the Problem

First and foremost, the sample must be self-consistent: a state in the positive sample cannot verify a CTL formula while another bisimilar state in the negative sample does not.

Definition 6 (Sample). *Given a KS $\mathcal{K} = (Q, \delta, \lambda)$, a sample of \mathcal{K} is a pair $(S^+, S^-) \in 2^Q \times 2^Q$ such that $\forall q^+ \in S^+, \forall q^- \in S^-, q^+ \not\sim_c q^-$.*

We define the *characteristic number* $\mathcal{C}_{\mathcal{K}}(S^+, S^-)$ of a sample as the smallest integer $c \in \mathbb{N}$ such that for every $q^+ \in S^+, q^- \in S^-, q^+ \not\sim_c q^-$.

Definition 7 (Consistent Formula). *A CTL formula φ is said to be consistent with a sample (S^+, S^-) of \mathcal{K} if $\forall q^+ \in S^+, q^+ \models_{\mathcal{K}} \varphi$ and $\forall q^- \in S^-, q^- \not\models_{\mathcal{K}} \varphi$.*

The rest of our article focuses on the following passive learning problems:

Definition 8 (Learning Problem). *Given a sample (S^+, S^-) of a KS \mathcal{K} and $n \in \mathbb{N}^*$, we introduce the following instances of the CTL learning problem:*

$L_{CTL(E)}(\mathcal{K}, S^+, S^-)$. *Is there $\varphi \in CTL(E)$ consistent with (S^+, S^-) ?*

$L_{CTL(E)}^{\leq n}(\mathcal{K}, S^+, S^-)$. *Is there $\varphi \in CTL(E)$, $|\varphi| \leq n$, consistent with (S^+, S^-) ?*

$ML_{CTL(E)}(\mathcal{K}, S^+, S^-)$. *Find the smallest $\varphi \in CTL(E)$ consistent with (S^+, S^-) .*

Theorem 4. $L_{CTL}(\mathcal{K}, S^+, S^-)$ and $ML_{CTL}(\mathcal{K}, S^+, S^-)$ always admit a solution.

Proof. Consider $\psi = \bigvee_{q^+ \in S^+} \varphi_{q^+}$. This formula ψ is consistent with (S^+, S^-) by design. Thus $L_{CTL}(\mathcal{K}, S^+, S^-)$ always admits a solution, and so does the problem $ML_{CTL}(\mathcal{K}, S^+, S^-)$, although ψ is unlikely to be the minimal solution. \square

Bordais et al. [5] proved that $L_{CTL}^{\leq n}(\mathcal{K}, S^+, S^-)$ is NP-hard, assuming the set of atomic propositions AP is given as an input as well.

3.2 An Explicit Solution

We must find a formula consistent with the sample (S^+, S^-) , an easier problem than Browne et al. [6]’s answer to Theorem 2 that subsumes bisimilarity with an entire KS. As we know that every state in S^- is dissimilar to every state in S^+ , we will try to encode this fact in CTL form, then use said encoding to design a formula consistent with the sample.

Definition 9 (Separating Formula). *Let (S^+, S^-) be a sample of a KS $\mathcal{K} = (Q, \delta, \lambda)$. Assuming that AP and Q are ordered, and given $q_1, q_2 \in Q$ such that $q_1 \not\sim q_2$, formula D_{q_1, q_2} is defined inductively w.r.t. $c = \mathcal{C}_{\mathcal{K}}(\{q_1\}, \{q_2\})$ as follows:*

- if $c = 0$ and $\lambda(q_1) \setminus \lambda(q_2) \neq \emptyset$ has minimal element a , then $D_{q_1, q_2} = a$;
- else if $c = 0$ and $\lambda(q_2) \setminus \lambda(q_1) \neq \emptyset$ has minimal element a , then $D_{q_1, q_2} = \neg a$;
- else if $c \neq 0$ and $\exists q'_1 \in \delta(q_1), \forall q'_2 \in \delta(q_2), q'_1 \not\sim_{c-1} q'_2$, then $D_{q_1, q_2} = \exists X \left(\bigwedge_{q'_2 \in \delta(q_2)} D_{q'_1, q'_2} \right)$, picking the smallest q'_1 verifying this property;
- else if $c \neq 0$ and $\exists q'_2 \in \delta(q_2), \forall q'_1 \in \delta(q_1), q'_1 \not\sim_{c-1} q'_2$, then $D_{q_1, q_2} = \forall X \neg \left(\bigwedge_{q'_1 \in \delta(q_1)} D_{q'_2, q'_1} \right)$, picking the smallest q'_2 verifying this property.

The formula $\mathcal{S}_{\mathcal{K}}(S^+, S^-) = \bigvee_{q^+ \in S^+} \bigwedge_{q^- \in S^-} D_{q^+, q^-} \in \text{CTL}(\{\neg, \wedge, \vee, \forall X, \exists X\})$ is then called the separating formula of sample (S^+, S^-) .

Intuitively, the CTL formula D_{q_1, q_2} merely expresses that states q_1 and q_2 are dissimilar by negating Definition 2; it is such that $q_1 \models_{\mathcal{K}} D_{q_1, q_2}$ but $q_2 \not\models_{\mathcal{K}} D_{q_1, q_2}$. Either q_1 and q_2 have different labels, q_1 admits a successor that is dissimilar to q_2 ’s successors, or q_2 admits a successor that is dissimilar to q_1 ’s. The following result is proven in Appendix B:

Theorem 5. *The separating formula $\mathcal{S}_{\mathcal{K}}(S^+, S^-)$ is consistent with (S^+, S^-) .*

As proven in Appendix C, we can bound the size of $\mathcal{S}_{\mathcal{K}}(S^+, S^-)$:

Corollary 1. *Assume the KS \mathcal{K} has degree k and $c = \mathcal{C}_{\mathcal{K}}(S^+, S^-)$, then:*

- if $k \geq 2$, then $|\mathcal{S}_{\mathcal{K}}(S^+, S^-)| \leq (5 \cdot k^c + 1) \cdot |S^+| \cdot |S^-|$;
- if $k = 1$, then $|\mathcal{S}_{\mathcal{K}}(S^+, S^-)| \leq (2 \cdot c + 3) \cdot |S^+| \cdot |S^-|$.

4 SAT-Based Learning

The *universal* fragment $\text{CTL}_{\forall} = \text{CTL}(\{\neg, \wedge, \vee, \forall X, \forall F, \forall G, \forall U\})$ of CTL happens to be as expressive as the full logic [1, Def. 6.13]. For that reason, we will reduce a learning instance of CTL_{\forall} of rank n to an instance of the SAT problem. A similar reduction has been independently found by Roy et al. [22].

Lemma 1. *There exists a Boolean propositional formula Φ_n such that the instance $L_{\text{CTL}_{\forall}}^{\leq n}(\mathcal{K}, S^+, S^-)$ of the learning problem admits a solution φ if and only if the formula Φ_n is satisfiable.*

4.1 Modelling the Formula

Assume that there exists a syntactic DAG \mathcal{D} of size smaller than or equal to n representing the desired CTL formula φ . Let us index \mathcal{D} 's nodes in $[1 \dots n]$ in such a fashion that each node has a higher index than its children, as shown in Fig. 1. Hence, n always labels a root and 1 always labels a leaf.

Let $\mathcal{L} = \text{AP} \cup \{\top, \neg, \wedge, \vee, \forall X, \forall F, \forall G, \forall U\}$ be the set of labels that decorates the DAG's nodes. For each $i \in [1 \dots n]$ and $o \in \mathcal{L}$, we introduce a Boolean variable τ_i^o such that $\tau_i^o = 1$ if and only if the node of index i is labelled by o .

For all $i \in [1 \dots n]$ and $j \in [0 \dots i - 1]$, we also introduce a Boolean variable $l_{i,j}$ (resp. $r_{i,j}$) such that $l_{i,j} = 1$ (resp. $r_{i,j} = 1$) if and only if j is the left (resp. right) child of i . Having a child of index 0 stands for having no child at all in the actual syntactic DAG \mathcal{D} .

Three mutual exclusion clauses guarantee that each node of the syntactic DAG has exactly one label and at most one left child and one right child. Moreover, three other clauses ensure that a node labelled by an operator of arity x has exactly x actual children (by convention, if $x = 1$ then its child is to the right). These simple clauses are similar to Neider et al.'s encoding [16] and for that reason are not detailed here.

4.2 Applying the Formula to the Sample

For all $i \in [1 \dots n]$ and $q \in Q$, we introduce a Boolean variable φ_i^q such that $\varphi_i^q = 1$ if and only if state q verifies the sub-formula φ_i rooted in node i . The next clauses implement the semantics of the true symbol \top , the atomic propositions, and the CTL operator $\forall X$.

$$\bigwedge_{\substack{i \in [1 \dots n] \\ q \in Q}} (\tau_i^\top \implies \varphi_i^q) \quad (\text{sem}_\top)$$

$$\bigwedge_{i \in [1 \dots n]} \left[\left(\bigwedge_{a \in \lambda(q)} (\tau_i^a \implies \varphi_i^q) \right) \wedge \left(\bigwedge_{a \notin \lambda(q)} (\tau_i^a \implies \neg \varphi_i^q) \right) \right] \quad (\text{sem}_a)$$

$$\bigwedge_{\substack{i \in [2 \dots n] \\ k \in [1 \dots i-1]}} \left[(\tau_i^{\forall X} \wedge r_{i,k}) \implies \bigwedge_{q \in Q} \left(\varphi_i^q \iff \bigwedge_{q' \in \delta(q)} \varphi_k^{q'} \right) \right] \quad (\text{sem}_{\forall X})$$

Semantic clauses are structured as follows: an antecedent stating node i 's label and its possible children implies a consequent expressing φ_i^q 's semantics for each $q \in Q$. Clause **sem $_\top$** states that $q \models \top$ is always true; clause **sem $_a$** , that $q \models a$ if and only if q is labelled by a ; and clause **sem $_{\forall X}$** , that $q \models \forall X \psi$ if and only if all of q 's successors verify ψ . Similar straightforward clauses encode the semantics of the Boolean connectors \neg , \wedge and \vee .

CTL semantics are also characterized by fixed points, whose naive encoding might however capture spurious (least vs greatest) sets: we resort to the bounded

semantics $\forall F^u$, $\forall U^u$ and $\forall G^u$. For all $i \in [1 \dots n]$, $q \in Q$, and $u \in [0 \dots \alpha(q)]$, we introduce a Boolean variable $\rho_{i,q}^u$ such that $\rho_{i,q}^u = 1$ if and only if q verifies the sub-formula rooted in i according to the CTL bounded semantics of rank u (e.g. $q \models \forall F^u \psi$, assuming sub-formula $\forall F \psi$ is rooted in node i).

Thanks to Theorem 3 we can introduce the following equivalence clause:

$$\bigwedge_{i \in [2 \dots n]} \left[\left(\bigvee_{o \in \{\forall F, \forall G, \forall U\}} \tau_i^o \right) \implies \bigwedge_{q \in Q} (\varphi_i^q \iff \rho_{i,q}^{\alpha(q)}) \right] \quad (\text{sem}_\rho)$$

Property 3 yields two other clauses whose inclusion is not mandatory (they were left out by Roy et al. [22]) that further constrains the bounded semantics:

$$\bigwedge_{i \in [2 \dots n]} \left[(\tau_i^{\forall F} \vee \tau_i^{\forall U}) \implies \bigwedge_{\substack{q \in Q \\ u \in [1 \dots \alpha(q)]}} (\rho_{i,q}^{u-1} \implies \rho_{i,q}^u) \right] \quad (\text{ascent}_\rho)$$

$$\bigwedge_{i \in [2 \dots n]} \left[\tau_i^{\forall G} \implies \bigwedge_{\substack{q \in Q \\ u \in [1 \dots \alpha(q)]}} (\rho_{i,q}^u \implies \rho_{i,q}^{u-1}) \right] \quad (\text{descent}_\rho)$$

The next clause enables variable $\rho_{i,q}^u$ for temporal operators only:

$$\bigwedge_{i \in [2 \dots n]} \left[\left(\bigwedge_{o \in \{\forall F, \forall G, \forall U\}} \neg \tau_i^o \right) \implies \bigwedge_{\substack{q \in Q \\ u \in [0 \dots \alpha(q)]}} \neg \rho_{i,q}^u \right] \quad (\text{no}_\rho)$$

Properties 1 and 2 yield an inductive definition of bounded semantics. We only explicit the base case base_ρ and the semantics $\text{sem}_{\forall U}$ of $\forall U^u$, but also implement semantic clauses for the temporal operators $\forall F$ and $\forall G$.

$$\bigwedge_{\substack{i \in [2 \dots n] \\ k \in [1 \dots i-1]}} \left[\left(\left[\bigvee_{o \in \{\forall F, \forall G, \forall U\}} \tau_i^o \right] \wedge r_{i,k} \right) \implies \bigwedge_{q \in Q} (\rho_{i,q}^0 \iff \varphi_k^q) \right] \quad (\text{base}_\rho)$$

$$\bigwedge_{\substack{i \in [2 \dots n] \\ j, k \in [1 \dots i-1]}} \left[(\tau_i^{\forall U} \wedge l_{i,j} \wedge r_{i,k}) \implies \bigwedge_{\substack{q \in Q \\ u \in [1 \dots \alpha(q)]}} \left(\rho_{i,q}^u \iff \left[\varphi_j^q \vee \left(\varphi_k^q \wedge \bigwedge_{q' \in \delta(q)} \rho_{i,q'}^{\min(\alpha(q'), u-1)} \right) \right] \right) \right] \quad (\text{sem}_{\forall U})$$

Finally, the last clause ensures that the full formula φ (rooted in node n) is verified by the positive sample but not by the negative sample.

$$\left(\bigwedge_{q^+ \in S^+} \varphi_n^{q^+} \right) \wedge \left(\bigwedge_{q^- \in S^-} \neg \varphi_n^{q^-} \right) \quad (\text{sem}_\varphi)$$

4.3 Solving the SAT Instance

We finally define the formula Φ_n as the conjunction of all the aforementioned clauses. Assuming an upper bound d on the KS's recurrence diameter, this encoding requires $\mathcal{O}(n^2 + n \cdot |\text{AP}| + n \cdot |Q| \cdot d)$ variables and $\mathcal{O}(n \cdot |\text{AP}| + n^3 \cdot |Q| \cdot d + n \cdot |\text{AP}| \cdot |Q|)$ clauses, not taking transformation to conjunctive normal form into account. By design, Lemma 1 holds.

Proof. The syntactic clauses allow one to infer the DAG of a formula $\varphi \in \text{CTL}$ of size smaller than or equal to n from the valuations taken by the variables (τ_i^o) , $(l_{i,j})$, and $(r_{i,j})$. Clauses **sem_a** to **sem_ϕ** guarantee that the sample is consistent with said formula φ , thanks to Theorem 3 and Properties 1, 2, and 3. \square

5 Algorithms for the Minimal Learning Problem

We introduce in this section an algorithm to solve the minimum learning problem $\text{ML}_{\text{CTL}_\forall}(\mathcal{K}, S^+, S^-)$. Remember that it always admits a solution if and only if the state sample is consistent by Theorem 4.

5.1 A Bottom-Up Algorithm

By Theorem 4, there exists a rank n_0 such that the problem $\text{L}_{\text{CTL}_\forall}^{\leq n_0}(\mathcal{K}, S)$ admits a solution. Starting from $n = 0$, we can therefore try to solve $\text{L}_{\text{CTL}_\forall}^{\leq n}(\mathcal{K}, S)$ incrementally until a (minimal) solution is found, in a similar manner to Neider and Gavran [16]. Algorithm 1 terminates with an upper bound n_0 on the number of required iterations.

Input: a KS \mathcal{K} and a sample S .
Output: the smallest CTL_\forall formula φ consistent with S .
 $n \leftarrow 0$;
repeat
 $n \leftarrow n + 1$;
 compute Φ_n ;
until Φ_n is satisfiable by some valuation v ;
from v build and **return** φ .

Algorithm 1: Solving $\text{ML}_{\text{CTL}_\forall}(\mathcal{K}, S)$.

5.2 Embedding Negations

The CTL formula $\exists F a$ is equivalent to the CTL_\forall formula $\neg \forall G \neg a$, yet the former remains more succinct, being of size 2 instead of 4. While CTL_\forall has been proven to be as expressive as CTL, the sheer amount of negations needed to express an equivalent formula can significantly burden the syntactic DAG. A possible optimization is to no longer consider the negation \neg as an independent operator but instead embed it in the nodes of the syntactic DAG, as shown in Fig. 2.

Note that such a definition of the syntactic DAG alters one's interpretation of a CTL formula's size: as a consequence, under this optimization, Algorithm 1

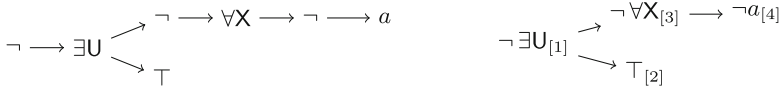


Fig. 2. The syntactic DAG of $\neg\exists\top U\neg\forall X\neg a$, before and after embedding negations.

may yield a formula with many negations that is no longer minimal under the original definition of size outlined in Sect. 2.2.

Formally, for each $i \in [1 \dots n]$, we introduce a new variable ν_i such that $\nu_i = 0$ if and only if the node of index i is negated. As an example, in Fig. 2, $\nu_1 = \nu_3 = \nu_4 = 0$, but $\nu_2 = 1$ and the sub-formula rooted in node 3 is $\neg\forall X\neg a$.

We then change the SAT encoding of CTL_{\forall} 's semantics accordingly. We remove the \neg operator from the syntactic DAG clauses and the set \mathcal{L} of labels. We delete its semantics and update the semantic clauses of the other operators. Indeed, the right side of each equivalence expresses the semantics of the operator rooted in node i before applying the embedded negation; we must therefore change the left side of the semantic equivalence accordingly, replacing the Boolean variable φ_i^q with the formula $\tilde{\varphi}_i^q = (\neg\nu_i \wedge \neg\varphi_i^q) \vee (\nu_i \wedge \varphi_i^q)$ that is equivalent to φ_i^q if $\nu_i = 1$ and $\neg\varphi_i^q$ if $\nu_i = 0$.

5.3 Optimizations and Alternatives

Minimizing the Input KS. In order to guarantee that an input S is indeed a valid sample, one has to ensure no state in the positive sample is bisimilar to a state in the negative sample. To do so, one has to at least partially compute the bisimilarity relation \sim on $\mathcal{K} = (Q, \delta, \lambda)$. But refining it to completion can be efficiently performed in $\mathcal{O}(|Q| \cdot |\text{AP}| + |\delta| \cdot \log(|Q|))$ operations [1, Thm. 7.41], yielding a bisimilar KS \mathcal{K}_{\min} of minimal size.

Minimizing the input KS is advantageous as the size of the semantic clauses depends on the size of \mathcal{K} , and the SAT solving step is likely to be the computational bottleneck. As a consequence, we always fully compute the bisimulation relation \sim on \mathcal{K} and minimize said KS.

Approximating the Recurrence Diameter. Computing the recurrence diameter of a state q is unfortunately an NP-hard problem that is known to be hard to approximate [4]. A coarse upper bound is $\alpha(q) \leq |Q| - 1$: it may however result in a significant number of unnecessary variables and clauses. Fortunately, the decomposition of a KS \mathcal{K} into strongly connected components (SCCs) yields a finer over-approximation shown in Fig. 3 that relies on the ease of computing α in a DAG. It is also more generic and suitable to CTL than existing approximations dedicated to LTL bounded model checking [14].

Contracting each SCC to a single vertex yields a DAG known as the *condensation* of \mathcal{K} . We weight each vertex of this DAG \mathcal{G} with the number of vertices in the matching SCC. Then, to each state q in the original KS \mathcal{K} , we associate the weight $\beta(q)$ of the longest path in the DAG \mathcal{G} starting from q 's SCC, minus one

(in order not to count q). Intuitively, our approximation assumes that a simple path entering a SCC can always visit every single one of its states once before exiting, a property that obviously does not hold for two of the SCCs shown here.

Encoding the Full Logic. CTL_{\forall} is semantically exhaustive but the existential temporal operators commonly appear in the literature; we can therefore consider the learning problem on the full CTL logic by integrating the operators $\exists X$, $\exists F$, $\exists G$, and $\exists U$ and provide a Boolean encoding of their semantics. We also consider the fragment $\text{CTL}_{\cup} = \{\neg, \vee, \exists X, \exists G, \exists U\}$ used by Roy et al. [22].

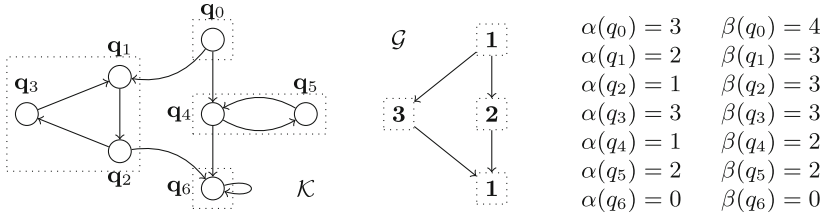


Fig. 3. An approximation β of the recurrence diameter α relying on SCC decomposition that improves upon the coarse upper bound $\alpha(q) \leq |Q| - 1 = 6$.

6 Experimental Implementation

We implement our learning algorithm in a C++ prototype tool **LearnCTL**¹ relying on Microsoft’s **Z3** due to its convenient C++ API. It takes as an input a sample of positive and negative KSs with initial states, then coalesced into a single KS and a sample of states compatible with the theoretical framework we described. It finally returns a separating CTL_{\forall} , CTL , or CTL_{\cup} formula after a sanity check performed by model-checking the input KSs against the learnt formula, using a simple algorithm based on Theorem 3.

6.1 Benchmark Collection

We intend on using our tool to generate formulas that can explain flaws in faulty implementations of known protocols. To do so, we consider structures generated by higher formalisms such as program graphs: a single mutation in the program graph results in several changes in the resulting KS. This process has been achieved manually according to the following criteria:

- The mutations only consist in deleting lines.
- The resulting KS should be *small*, less than ~ 1000 states.
- Any mutation should result in a syntactically correct model.

¹ publicly available at <https://gitlab.lre.epita.fr/adrien/learnctl>.

We collected program graphs in a toy specification language for a CTL model checker class implemented in Java. Furthermore, we also considered PROMELA models from the Spin model-checker [12] repository. Translations were then performed through the Python interface of spot/ltsmin [9, 13].

Example 1. Consider the mutual exclusion protocol proposed by [18] and specified in PROMELA in Fig. 4 that generates a KS with 55 states. We generate mutants by deleting no more than one line of code at a time, ignoring variable and process declarations as they are necessary for the model to be compiled and the two assertion lines that are discarded by our KS generator, our reasoning being that subtle changes yield richer distinguishing formulas.

Furthermore, removing the instruction `ncrit--` alone would lead to an infinite state space; thus, its deletion is only considered together with the instruction `ncrit++`. Finally, we set some atomic propositions of interest: *c* stands for at least one process being in the critical section (`ncrit>0`), *m* for both processes (`ncrit>1`), and *t* for process 0’s turn. An extra *dead* atomic proposition is added by Spot/LTSMIn to represent deadlocked states.

As summarized on Fig. 4, every mutated model, once compared with the original KS, lead to distinguishing formulas characterizing Peterson’s protocol: mutations `m1`, `m2`, and `m3` yield a mutual exclusion property, `m4` yields a liveness property, `m5` yields a fairness property, and `m6` yields global liveness formula.

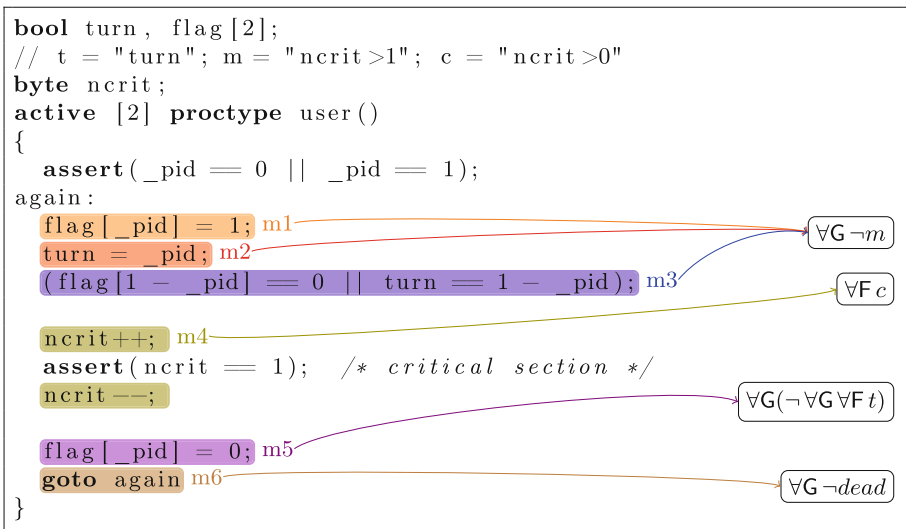


Fig. 4. Peterson’s mutual exclusion protocol in PROMELA and learnt formulas for each deleted instruction.

6.2 Quantitative Evaluation

We quantitatively assess the performance of the various optimizations and CTL fragments discussed previously. To do so, we further enrich the benchmark series through the use of random mutations of hard-coded KSs: these mutations may alter some states, re-route some edges, and spawn new states. We consider a total of 234 test samples, ranging from size 11 to 698 after minimization. We perform the benchmarks on a GNU/Linux Debian machine (*bullseye*) with 24 cores (Intel(R) Xeon(R) CPU E5-2620 @ 2.00 GHz) and 256Go of RAM, using version 4.8.10 of *libz3* and 1.0 of *LearnCTL*.

Table 1 displays a summary of these benchmarks: β stands for the refined approximation of the recurrence diameter described in Sect. 5.3; \neg , for the embedding of negations in the syntactic tree introduced in Sect. 5.2. The average size of the syntactic DAGs learnt is 4.14.

Option β yields the greatest improvement, being on average at least 6 times faster than the default configuration; option \neg further divides the average runtime by at least 2. These two optimizations alone speed up the average runtime by a factor of 12 to 20. The CTL fragment used, all other options being equal, does not influence the average runtime as much (less than twofold in the worst case scenario); $(\text{CTL}_U, \beta, \neg)$ is the fastest option, closely followed by $(\text{CTL}_V, \beta, \neg)$.

Intuitively, approximating the recurrence diameter aggressively cuts down the number of SAT variables needed: assuming that α has upper bound d , we only need $n \cdot |Q| \cdot d$ Boolean variables ($\rho_{i,q}^u$) instead of $n \cdot |Q|^2$. Moreover, embedding negations, despite requiring more complex clauses, results in smaller syntactic DAGs with “free” negations, hence faster computations, keeping in mind that the last SAT instances are the most expensive to solve, being the largest.

	-	β	β, \neg
CTL_V	50 46870	14 6493	4 2271
CTL	50 42658	8 5357	5 3370
CTL_U	46 31975	28 5064	4 1987

Table 1. Number of timeouts at ten minutes | arithmetic mean (in milliseconds) on the 178 samples that never timed out of various options and fragments.

Unsurprisingly, $(\text{CTL}_V, \beta, \neg)$ and $(\text{CTL}, \beta, \neg)$ outperform $(\text{CTL}_U, \beta, \neg)$ when a minimal distinguishing formula using the operator $\forall U$ exists: the duality between $\forall U$ and $\exists U$ is complex and, unlike the other operators, cannot be handled at no cost by the embedded negation as it depends on the *release* operator.

Figure 5 further displays a log-log plot comparing the runtime of the most relevant fragments and options to $(\text{CTL}_U, \beta, \neg)$. For a given set of parameters, each point stands for one of the 234 test samples. Points above the black diagonal favour $(\text{CTL}_U, \beta, \neg)$; points below, the aforementioned option. Points on the second dotted lines at the edges of the figure represent timeouts.

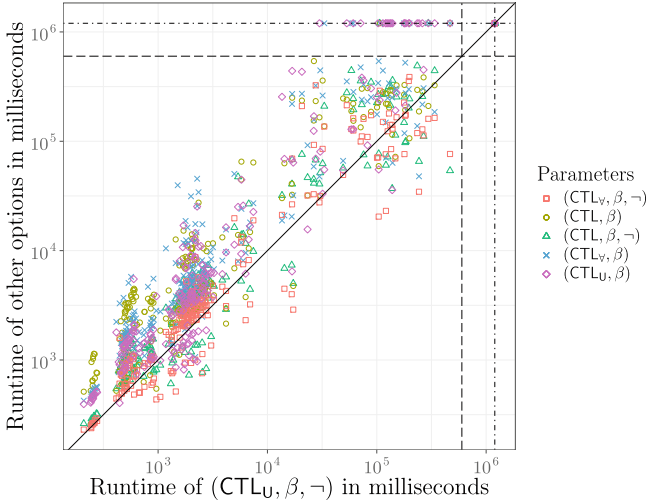


Fig. 5. Comparing (CTL_U, β, \neg) to other options on every sample.

7 Conclusion and Further Developments

We explored in this article the CTL learning problem: we first provided a direct explicit construction before relying on a SAT encoding inspired by bounded model-checking to iteratively find a minimal answer. We also introduced in Sect. 3 an explicit answer to the learning problem that belongs to the fragment $CTL(\neg, \wedge, \vee, \forall X, \exists X)$. It remains to be seen if a smaller formula can be computed using a more exhaustive selection of CTL operators. A finer grained explicit solution could allow one to experiment with a top-down approach as well.

Moreover, we provided a dedicated C++ implementation, and evaluated it on models of higher-level formalisms such as PROMELA. Since the resulting KSS have large state spaces, further symbolic approaches are to be considered for future work, when dealing with program graphs instead of Kripke structures. In this setting, one might also consider the synthesis problem of the relevant atomic propositions from the exposed program variables. Nevertheless, the experiments on Kripke structures already showcase the benefits of the approximated recurrence diameter computation and of our extension of the syntactic DAG definition, as well as the limited relevance of the target CTL fragment.

Another avenue for optimizations can be inferred from the existing SAT-based LTL learning literature: in particular, Rienier et al. [20] relied on a topology-guided approach by explicitly enumerating the possible shapes of the syntactic DAG and solving the associated SAT instances in parallel. Given the small size on average of the formulas learnt so far and the quadratic factor impacting the number of semantic clauses such as $sem_{\forall U}$ due to the structural variables $l_{i,j}$ and $r_{i,k}$, this approach could yield huge performance gains in CTL’s case as well.

We relied on Z3’s convenient C++ API, but intuit that we would achieve better performance with state-of-the-art SAT solvers such as the winners of the yearly SAT competition [2]. We plan on converting our Boolean encoding to the DIMACS CNF format in order to interface our tool with modern SAT solvers.

Finally, it is known that the bounded learning problem is NP-complete, but we would also like to find the exact complexity class of the minimal CTL learning problem. We intuit that it is not, Kripke structures being a denser encoding in terms of information than lists of linear traces: as an example, one can trivially compute an LTL formula (resp. a CTL formula) of polynomial size that distinguishes a sample of ultimately periodic words (resp. of finite computation trees with lasso-shaped leaves), but the same cannot be said of a sample of Kripke structures. It remains to be seen if this intuition can be confirmed or infirmed by a formal proof.

A Proof of Theorem 3

$\forall F$. Assume that $q \models \forall F \varphi$. Let us prove that $q \models \forall F^{\alpha(q)} \varphi$. Consider a run $r = (s_i) \in \mathcal{R}(q)$. By hypothesis, we can define the index $j = \min\{i \in \mathbb{N} \mid s_i \models \varphi\}$.

Now, assume that $j > \alpha(q)$. By definition of the recurrence diameter α , $\exists k_1, k_2 \in [0 .. j - 1]$ such that $k_1 \leq k_2$ and $s_{k_1} = s_{k_2}$. Consider the finite runs $u = (s_i)_{i \in [0..k_1]}$ and $v = (s_i)_{i \in [k_1+1..k_2]}$. We define the infinite, ultimately periodic run $r' = u \cdot v^\omega = (s'_i)$. By definition of j , $\forall i \in \mathbb{N}$, $s'_i \not\models \varphi$ in order to preserve the minimality of j . Yet $r' \in \mathcal{R}(q)$ and $q \models \forall F \varphi$. By contradiction, $j \leq \alpha(q)$. As consequence, $(q \models \forall F \varphi) \implies (q \models \forall F^{\alpha(q)} \varphi)$ holds.

Trivially, $(q \models \forall F^{\alpha(q)} \varphi) \implies (q \models \forall F \varphi)$ holds. Hence, we have proven both sides of the desired equivalence for $\forall F$.

$\forall G$. Assume that $q \models \forall G^{\alpha(q)} \varphi$. Let us prove that $q \models \forall G \varphi$. Consider a run $r = (s_i) \in \mathcal{R}(q)$ and $j \in \mathbb{N}$. Let us prove that $s_j \models \varphi$.

State s_j is obviously reachable from q . Let us consider a finite run without repetition $u = (s'_i)_{i \in [0..k]}$ such that $s_0 = q$ and $s'_k = s_j$. By definition of the recurrence diameter, $k \leq \alpha(q)$. We define the infinite runs $v = (s_i)_{i > j}$ and $r' = u \cdot v$. Since $r' \in \mathcal{R}(q)$ and $q \models \forall G^{\alpha(q)} \varphi$, $s_k \models \varphi$, hence $s_j \models \varphi$. As a consequence, $(q \models \forall G^{\alpha(q)} \varphi) \implies (q \models \forall G \varphi)$.

Trivially, $(q \models \forall G \varphi) \implies (q \models \forall G^{\alpha(q)} \varphi)$ holds. Hence, we have proven both sides of the desired equivalence for $\forall G$.

$\exists F$ and $\exists G$. Formula $\exists F \varphi$ (rep. $\exists G \varphi$) being equivalent to the dual formula $\neg \forall G \neg \varphi$ (resp. $\neg \forall F \neg \varphi$), the previous proofs immediately yield the desired equivalences.

$\forall U$ and $\exists U$. We can handle the case of $\forall \varphi U \psi$ in a manner similar to $\forall F$: we prove by contradiction that the first occurrence of ψ always happens in less than $\alpha(q)$ steps. And the semantic equivalence for $\exists \varphi U \psi$ can be handled in a fashion similar to $\forall G$: an existing infinite run yields a conforming finite prefix without repetition of length lesser than or equal to $\alpha(q)$.

B Proof of Theorem 5

Given two dissimilar states $q_1, q_2 \in Q$, let us prove by induction on the characteristic number $c_{q_1, q_2} = \mathcal{C}_{\mathcal{K}}(\{q_1\}, \{q_2\})$ that $q_1 \models D_{q_1, q_2}$ and $q_2 \not\models D_{q_1, q_2}$.

Base Case. If $c_{q_1, q_2} = 0$, then by definition $\lambda(q_1) \neq \lambda(q_2)$ and by design D_{q_1, q_2} is a literal (i.e. an atomic proposition or the negation thereof) verified by q_1 but not by q_2 .

Inductive Case. Assume that the property holds for all characteristic numbers smaller than or equal to $c \in \mathbb{N}$. Consider two states $q_1, q_2 \in Q$ such that $c_{q_1, q_2} = c + 1$. By definition of the refined equivalence relation \sim_{c+1} , $\exists q'_1 \in \delta(q_1), \forall q'_2 \in \delta(q_2), q'_1 \not\sim_c q'_2$, hence $c_{q'_1, q'_2} \leq c$.

By induction hypothesis, $D_{q'_1, q'_2}$ is well-defined, $q'_1 \models D_{q'_1, q'_2}$ and $q'_2 \not\models D_{q'_1, q'_2}$.

As a consequence, D_{q_1, q_2} is well-defined, $q_1 \models \exists X \left(\bigwedge_{q'_2 \in \delta(q_2)} D_{q'_1, q'_2} \right)$, and $q_2 \not\models$

$$\exists X \left(\bigwedge_{q'_2 \in \delta(q_2)} D_{q'_1, q'_2} \right).$$

We handle the case where $\exists q'_2 \in \delta(q_2), \forall q'_1 \in \delta(q_1), q'_1 \not\sim_c q'_2$ in a similar fashion. As a consequence, the property holds at rank $c + 1$.

Therefore, for each $q^+ \in S^+$ and each $q^- \in S^-$, $q^+ \models D_{q^+, q^-}$ and $q^- \not\models D_{q^+, q^-}$. Hence, $q^+ \models \mathcal{S}_{\mathcal{K}}(S^+, S^-)$ and $q^- \not\models \mathcal{S}_{\mathcal{K}}(S^+, S^-)$. \square

C Proof of Corollary 1

First, given $q^+ \in S^+$ and $q^- \in S^-$, let us bound the size of D_{q^+, q^-} based on their characteristic number $c_{q_1, q_2} = \mathcal{C}_{\mathcal{K}}(\{q_1\}, \{q_2\})$.

$$c_{q_1, q_2} = 0 \implies |D_{q^+, q^-}| \leq 2 \qquad \text{as } \lambda(q_1) \neq \lambda(q_2)$$

$$c_{q_1, q_2} \geq 1 \implies |D_{q^+, q^-}| \leq (k + 1) + \sum_{q'_2 \in \delta(q_2)} |D_{q'_1, q'_2}| \quad \text{for some } q'_1 \in \delta(q_1)$$

$$\text{or } |D_{q^+, q^-}| \leq (k + 1) + \sum_{q'_1 \in \delta(q_1)} |D_{q'_1, q'_2}| \quad \text{for some } q'_2 \in \delta(q_2)$$

We are looking for an upper bound $(U_n)_{n \geq 0}$ such that $\forall n \in \mathbb{N}, \forall q^+ \in S^+, \forall q^- \in S^-$, if $c_{q^+, q^-} \leq n$, then $|D_{q^+, q^-}| \leq U_n$. We define it inductively:

$$U_0 = 2$$

$$U_{n+1} = k \cdot U_n + k + 1$$

Assuming $k \geq 2$, we explicit the bound $U_n = (2 + \frac{k+1}{k-1}) \cdot k^n - \frac{k+1}{k-1} \leq 5 \cdot k^n$. As $(\{q^+\}, \{q^-\})$ is a sub-sample of S , $c_{q^+, q^-} \leq c$ and $|D_{q^+, q^-}| \leq U_c \leq 5 \cdot k^c$. We

can finally bound the size of $\mathcal{S}_{\mathcal{K}}(S^+, S^-)$:

$$\begin{aligned} |\mathcal{S}_{\mathcal{K}}(S^+, S^-)| &\leq (|S^+| - 1) \cdot (|S^-| - 1) + \sum_{\substack{q^+ \in S^+ \\ q^- \in S^-}} |D_{q^+, q^-}| \\ &\leq |S^+| \cdot |S^-| + |S^+| \cdot |S^-| \cdot U_c \\ &\leq (5 \cdot k^c + 1) \cdot |S^+| \cdot |S^-| \end{aligned}$$

Yielding the aforementioned upper bound. If $k = 1$, then $U_n = 2 \cdot n + 2$ and the rest of the proof is similar to the previous case. \square

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
2. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2023)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 193–207. Springer, Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
4. Björklund, A., Husfeldt, T., Khanna, S.: Approximating longest directed paths and cycles. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 222–233. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_21
5. Bordais, B., Neider, D., Roy, R.: Learning temporal properties is NP-hard (2023)
6. Browne, M., Clarke, E., Grumberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1), 115–131 (1988). [https://doi.org/10.1016/0304-3975\(88\)90098-9](https://doi.org/10.1016/0304-3975(88)90098-9)<https://www.sciencedirect.com/science/article/pii/0304397588900989>
7. Camacho, A., McClraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 29, no. 1, pp. 621–630 (May 2021). <https://doi.org/10.1609/icaps.v29i1.3529>, <https://ojs.aaai.org/index.php/ICAPS/article/view/3529>
8. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). <https://doi.org/10.1007/BFb0025774>
9. Duret-Lutz, A., et al.: From Spot 2.0 to Spot 2.10: What’s new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV’22). LNCS, vol. 13372, pp. 174–187. Springer (Aug 2022). https://doi.org/10.1007/978-3-031-13188-2_9
10. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: Chandler, J., Eyraud, R., Heinz, J., Jardine, A., van Zaanen, M. (eds.) Proceedings of the 15th International Conference on Grammatical Inference, 23–27 August 2021, Virtual Event. Proceedings of Machine Learning Research, vol. 153, pp. 237–250. PMLR (2021). <https://proceedings.mlr.press/v153/fijalkow21a.html>

11. Gaglione, J.R., Neider, D., Roy, R., Topcu, U., Xu, Z.: Learning linear temporal properties from noisy data: a MaxSAT-based approach. In: Hou, Z., Ganesh, V. (eds.) *Automated Technology for Verification and Analysis*, pp. 74–90. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_6
12. Holzmann, G.J.: *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley (2004)
13. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015. LNCS*, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
14. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) *Verification, Model Checking, and Abstract Interpretation*, pp. 298–309. Springer, Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-36384-X_24
15. Lutz, S., Neider, D., Roy, R.: Specification sketching for linear temporal logic. In: André, É., Sun, J. (eds.) *Automated Technology for Verification and Analysis*, pp. 26–48. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_2
16. Neider, D., Gavran, I.: Learning linear temporal properties. In: Bjørner, N.S., Gurfinkel, A. (eds.) *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pp. 1–10. IEEE (2018). <https://doi.org/10.23919/FMCAD.2018.8603016>, <https://doi.org/10.23919/FMCAD.2018.8603016>
17. Penczek, W., Woźna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. *Fundam. Inf.* **51**(1–2), 135–156 (2002)
18. Peterson, G.L.: Myths about the mutual exclusion problem. *Inf. Process. Lett.* **12**(3), 115–116 (1981). [https://doi.org/10.1016/0020-0190\(81\)90106-X](https://doi.org/10.1016/0020-0190(81)90106-X), [https://doi.org/10.1016/0020-0190\(81\)90106-X](https://doi.org/10.1016/0020-0190(81)90106-X)
19. Pnueli, A.: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57 (1977). <https://doi.org/10.1109/SFCS.1977.32>
20. Riemer, H.: Exact synthesis of LTL properties from traces. In: *2019 Forum for Specification and Design Languages (FDL)*, pp. 1–6 (2019). <https://doi.org/10.1109/FDL.2019.8876900>
21. Roy, R., Fisman, D., Neider, D.: Learning interpretable models in the property specification language. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence. IJCAI 2020* (2021)
22. Roy, R., Neider, D.: *Inferring properties in computation tree logic* (2023)
23. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 295–306 (Nov 2009). <https://doi.org/10.1109/ASE.2009.30>
24. Xu, L., Chen, W., Xu, Y.Y., Zhang, W.H.: Improved bounded model checking for the universal fragment of CTL. *J. Comput. Sci. Technol.* **24**(1), 96–109 (2009). <https://doi.org/10.1007/s11390-009-9208-5>, <https://doi.org/10.1007/s11390-009-9208-5>
25. Zhang, W.: Bounded semantics of CTL and sat-based verification. In: Breitman, K., Cavalcanti, A. (eds.) *Formal Methods and Software Engineering*, pp. 286–305. Springer, Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_15

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

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





MCSat-Based Finite Field Reasoning in the *YICES2* SMT Solver (Short Paper)

Thomas Hader¹✉, Daniela Kaufmann¹ , Ahmed Irfan² ,
Stéphane Graham-Lengrand² , and Laura Kovács¹

¹ TU Wien, Vienna, Austria

{thomas.hader,daniela.kaufmann,laura.kovacs}@tuwien.ac.at

² SRI International, Menlo Park, CA, USA

{ahmed.irfan,stephane.graham-lengrand}@sri.com

Abstract. This system description introduces an enhancement to the *YICES2* SMT solver, enabling it to reason over non-linear polynomial systems over finite fields. Our reasoning approach fits into the model-constructing satisfiability (MCSat) framework and is based on zero decomposition techniques, which find finite basis explanations for theory conflicts over finite fields. As the MCSat solver within *YICES2* can support (and combine) several theories via theory plugins, we implemented our reasoning approach as a new plugin for finite fields and extended *YICES2*'s frontend to parse finite field problems, making our implementation the first MCSat-based reasoning engine for finite fields. We present its evaluation on finite field benchmarks, comparing it against *CVC5*. Additionally, our work leverages the modular architecture of the MCSat solver in *YICES2* to provide a foundation for the rapid implementation of further reasoning techniques for this theory.

Keywords: SMT solving · MCSat · finite fields · polynomial arithmetic

1 Introduction

Satisfiability Modulo Theories (SMT) solving plays a crucial role in automated reasoning as it combines the power of Boolean satisfiability (SAT) with various mathematical background theories [3]. This connection enables the automated verification and synthesis of systems [15] that require reasoning in more expressive logical theories, for example real/integer arithmetic.

State-of-the-art SMT solvers employ a combination of Boolean level reasoning and theory-specific algorithms. This is achieved either through the use of the CDCL(T) paradigm [16] or the model-constructing satisfiability (MCSat) approach [11, 14]. The MCSat algorithm lifts the Boolean-level CDCL algorithm to the theory level, while keeping the search theory independent. This approach is particularly effective for handling complex arithmetic theories. For instance, *YICES2* [5] uses the MCSat approach to handle non-linear arithmetic constraints.

Finite fields offer an ideal framework for modeling bounded machine arithmetic, particularly relevant in the context of contemporary cryptosystems utilized in system security and post-quantum cryptography. Current methodologies, for instance, develop private and secure systems using zero-knowledge (ZK) proofs [7] or authenticate blockchain technologies like smart contracts [19]. Verifying applications in such areas require efficient SMT solvers that support reasoning over finite field arithmetic, e.g., verification of a compiler for ZK proofs [18].

Related Work. Currently, the related work on SMT solving in finite field arithmetic is still rather limited. Our own theoretical work [9] on MCSat approaches based on finding zero decompositions comes with a proof-of-concept implementation that facilitates only a very fundamental MCSat algorithm, has only limited support of Boolean propagation, and is unable to parse SMT-LIB 2 [2].

The only other SMT solver that we are aware of being capable of reasoning over finite fields is CVC5 [1, 17], which uses a classical CDCL(T) approach. As a theory engine, Gröbner bases [4] reasoning over a set of polynomial equalities is applied. If the derived Gröbner basis contains the constant 1, then the system is unsatisfiable and a conflict core for the CDCL(T) search can be found. Otherwise, a guided enumeration of all possible solutions is performed to search for a model.

Note that both approaches [9, 17] use complementary techniques. On the one hand, Gröbner bases are highly engineered to find conflicts in the polynomial input, which tends to help for unsatisfiable instances [17]. On the other hand, a model constructing approach tends to be fast whenever there is a solution, especially when there is a high number of models [9].

We further note that at the moment our implementation in YICES2, as well as CVC5, is restricted to finite fields where the order (i.e. size) is prime. This limitation is sufficient for many applications in cryptosystems and ZK proofs.

Besides using dedicated finite field solvers, problems over prime fields can be encoded in integer arithmetic using the modulo operator. Further, since terms are bounded, encoding as bit-vectors for subsequent bit-blasting is possible. However, prior experiments have shown that those encodings perform poorly on existing solvers [17].

Contributions. We present an integration of the theory of non-linear finite field arithmetic in the YICES2 SMT solver [5], enabling it to reason over finite field problems. This includes the following contributions which we will further explain throughout the rest of this paper:

- Adding the parsing of finite field problems to the YICES2 SMT-LIB 2 front-end (Sect. 3).
- Representing finite field polynomials as terms in YICES2 and implementing features regarding such polynomials in the LIBPOLY library [12], which is the library used for polynomials in YICES2 (Sect. 4).
- Implementing and evaluating an MCSat theory back-end for finite field reasoning, using existing concepts from non-linear real and bit-vector solving from YICES2 (Sect. 5).

To the best of our knowledge, our work is currently the only finite field instantiation of MCSat. While our initial theory reasoning approach follows closely the explanation generation procedure of our previous work [9], our implementation allows easy drop-in of an improved explanation procedure in the future.

2 Preliminaries

In mathematics, a finite field is a field that contains a finite number of elements. A finite field \mathbb{F}_p of prime order p can roughly be seen as the representation of the integers modulo the prime p . We refer to [9, 17] for details on the theory and representation of finite fields. Since there is no inherent order on finite fields, polynomial constraints are either equalities $p = 0$ or disequalities $p \neq 0$ for a finite field polynomial p .

For SMT solving in finite fields, we are interested in the following problem:

Given a finite field \mathbb{F}_p , where p is a prime number, let $X = x_1, \dots, x_n$, let F be a set of polynomial constraints in $\mathbb{F}_p[X]$ and \mathcal{F} a formula following the logical structure:

$$\mathcal{F} = \bigwedge_{C \subseteq F} \bigvee_{f \in C} f = \bigwedge_{C \subseteq F} \bigvee_{f \in C} \text{poly}(f) \triangleright 0 \quad \text{with } \triangleright \in \{=, \neq\}.$$

SMT solving over finite fields: Does an assignment $\nu : \{x_1, \dots, x_n\} \rightarrow \mathbb{F}_p^n$ exist that satisfies \mathcal{F} ?

For example, the formula $\mathcal{F}_1 = (x - 1 = 0 \vee y - 1 = 0) \wedge (xy - 1 = 0)$ is satisfied by the assignment $x \mapsto 1, y \mapsto 1$ in \mathbb{F}_3 ; whereas the formula $\mathcal{F}_2 = (x - 1 = 0 \vee y - 1 = 0) \wedge (xy - 1 = 0) \wedge (x - 2 = 0)$ is unsatisfiable in \mathbb{F}_3 .

Yices2 and MCSat. YICES2 contains two main solvers, one based on the traditional CDCL(T) approach [16] and one based on the MCSat approach [13, 14]. YICES2’s common API and front-ends can automatically select which solver to use at runtime, depending on an SMT-LIB 2 logic. In particular, when non-linear real or integer arithmetic constraints are present YICES2 selects the MCSat solver. The MCSat solver in YICES2 currently supports the theories of non-linear real arithmetic (QF_NRA) [13] and integer arithmetic (QF_NIA) [10], bit-vectors (QF_BV) [8], equality and uninterpreted functions (QF_EUF), arrays [6], and combinations thereof.

In contrast to CDCL(T) that *complements* CDCL with theory reasoning, MCSat applies CDCL-like mechanisms to *perform* theory reasoning. Specifically, it explicitly and incrementally constructs models with first-order variable assignments—maintained in a *trail*—while maintaining the invariant that none of the constraints evaluate to false. MCSat decides upon such assignments when there is choice, it can propagate them when there is not, and it backtracks upon conflict. The lemmas learned upon backtracking are based on theory-specific explanations of conflicts and propagations. This theory-specific reason-

ing is implemented through *plugins* that provide interfaces to make decisions, perform propagations, and produce explanations.

3 Usability of SMT Solving in Finite Fields

Support for finite field reasoning in YICES2 is available on the master branch¹ and will be included in the next release (2.7). The theory of finite fields can be accessed using a not-yet official extension of the SMT-LIB 2 language (.smt2).

SMT-LIB 2 Parsing. Extending the parser to handle finite field problems was our first extension to YICES2. Currently, polynomials over finite fields are no official theory in SMT-LIB 2 [2]. However, when implementing finite field support in CVC5 [1], an extension was proposed in [17]. In the interest of keeping inputs and benchmarks comparable, we aimed at a compatible implementation. Standardization efforts to create an official SMT-LIB 2 theory for finite field arithmetic are currently ongoing.

In the SMT-LIB 2 extension, the theory of (quantifier-free) non-linear finite field arithmetic is denoted as **QF_FFA**. Elements can be defined using the sort **FiniteField**. The sort is indexed by the order of the finite field, which is required to be a prime number. For instance `(_ FiniteField 3)` defines the finite field of order 3. Constants are indexed with the field order to indicate which finite field they belong in, e.g., `(_ ff2 3)`. Note that the integer following **ff** is interpreted modulo the field order. As a short-cut to avoid rewriting the field order over and over again, the **as** keyword can be used to interpret the constant in the correct field type: `(as ff2 FF3)` for a defined finite field sort **FF3**. To specify the finite field operations **ff.mul** and **ff.add** are available for multiplication and addition of finite field values, respectively. Both support an arbitrary number of operators. Atoms with finite field terms can be **=** with its respective meaning. For example, an encoding of \mathcal{F}_1 can be seen in Fig. 1.

```
(set-logic QF_FFA)
(define-sort FF3 () (_ FiniteField 3))
(declare-fun x () FF3)
(declare-fun y () FF3)
(assert (and
  (or (= (ff.add x (as ff-1 FF3)) (as ff0 FF3))
      (= (ff.add y (as ff-1 FF3)) (as ff0 FF3)))
  (= (ff.mul x y) (as ff-1 FF3)))
(check-sat)
```

Fig. 1. Example for an SMT-LIB 2 encoding of a finite field problem \mathcal{F}_1 .

¹ Available at <https://github.com/SRI-CSL/yices2>.

4 Implementation Details

The Implementation of MCSat in YICES2. The MCSat solver in YICES2 supports multiple theories via a notion of theory *plugin* that builds upon an earlier architecture [11]. An MCSat theory plugin in YICES2 implements a number of functionalities that are given to the main MCSat solver as function pointers. The main MCSat loop calls these functions for theory-specific operations such as deciding or propagating the value of variables or getting explanation lemmas, or upon certain events such as the creation of new terms and lemmas. In return, a theory plugin can access theory-generic mechanisms for, e.g., inspecting the MCSat trail, creating variables and requesting to be notified of certain events like variable assignments, as well as raising conflicts. A theory plugin is not required to implement mechanisms for propagating theory assignments and explaining them, but for the current theories in YICES2, such propagations have provided noticeable speed-ups (see, e.g., [8]).

The Finite Field MCSat Plugin. Before handling constraints in the finite field MCSat plugin, the input assertions are represented as polynomial constraints. Limited preprocessing (e.g., constant propagation) is performed at this step. Internally, the plugin only handles polynomial equalities and disequalities. The implementation of the finite field plugin follows an approach similar to the plugin for non-linear arithmetic [10].

Using the MCSat trail, the finite field plugin reads which constraints must be fulfilled at any given time (as decided or deduced by the Boolean plugin) and tracks the assignment of values to polynomial variables. It also tracks, for each polynomial variable, the *set of feasible values* that the variable can be assigned without any of the polynomial constraints evaluating to false: Using watch lists, it detects when any of the constraints becomes *unit*, i.e. when all of its variables but one have been assigned values. Upon such detection, it computes how the constraint restricts the set of feasible values for the last remaining variable, using regular univariate polynomial factorization. When that set becomes empty, the plugin reports a theory conflict to the main MCSat engine. Given a conflict core and the current assignment, the *explanation procedure* in the plugin generates a (globally valid) lemma that explains the conflict in that it excludes a class of assignments (including the current one) that all violate the conflict core. The MCSat engine performs conflict analysis using theory explanations and Boolean resolutions, and either backtracks if it can or concludes unsatisfiability. On the other hand, the instance is satisfiable once all variables are assigned a value.

Finite Field Explanations. In our earlier work [9], we presented an explanation procedure for finite fields. This approach employs subresultant regular subchains (SRS) [20] between conflicting polynomials to provide new polynomial constraints that can be propagated. In a nutshell, SRS can be used to construct a generalized greatest common divisor (GCD) of polynomials that takes into account the current partial variable assignment on the trail. The computed GCD is utilized in a zero decomposition procedure to reduce the degree of the

conflicting polynomials until we can learn a polynomial constraint that excludes the current partial assignment. This constraint is added as an explanation clause to resolve the conflict. We implemented the procedure of [9] in the current version of YICES2 using LIBPOLY. However, it is important to note that there are other solving techniques for polynomial systems over finite field that could potentially be utilized to develop an explanation method suitable for an MCSat-based search. Furthermore, it is still an open question how different techniques perform in an MCSat environment. That is why we have kept the explanation procedure encapsulated in our implementation, allowing for easy extension in order to support development and evaluation of future explanation procedures.

5 Evaluation

Since finite field solving is a rather new endeavor in the world of SMT, no extensive set of SMT-LIB 2 benchmarks exists yet. For the evaluation we have selected the benchmark sets presented in the papers describing the theory behind the implementation of YICES2 and CVC5:

- (i) The polynomial sets from our prior work [9], consisting of 325 instances. These benchmarks primarily contains finite fields up to order 211, using two classes of polynomial systems: *randomly generated* as well as *crafted systems*. The crafted benchmarks are product of (mostly) degree-1 polynomials.
- (ii) Benchmarks generated using ZK proof compilers presented in [17]. Besides polynomial equations, these 1602 benchmark instances also contain Boolean structure. The field order varies from small (11) up to vast (more than 2^{255}).

Experimental Setup. Our experiments were run on an AMD EPYC 7502 CPU with a timeout of 300 seconds per benchmark instance. We compared our implementation of YICES2 against CVC5 version 1.1.1, which is the latest released version at the time of writing. We are not aware of any further SMT solvers supporting the theory of non-linear finite fields to be included in the comparison.

Experimental Results. The performance comparison between the two solvers on the first benchmark set can be seen in Fig. 2 and Fig. 3 (left). It is clear to see that the random instances are harder to solve than the crafted instances (which have significantly more variables). We believe that this is due to the lack of internal structure in random polynomials. This makes symbolic handling of those polynomial systems hard, both for Gröbner basis computation (in CVC5) as well as SRS computation (in YICES2).

Note that CVC5 performs most symbolic computation upfront (when generating the Gröbner basis) and enumerates potential solutions in a second step (using auxiliary Gröbner basis calls). The MCSat approach in YICES2, on the other hand, interleaves model generation and symbolic computation during the search. This tends to be an advantage for harder polynomial systems especially together with small finite field orders. When the finite field order increases, this advantage seems to vanish. For the crafted polynomial benchmarks, YICES2

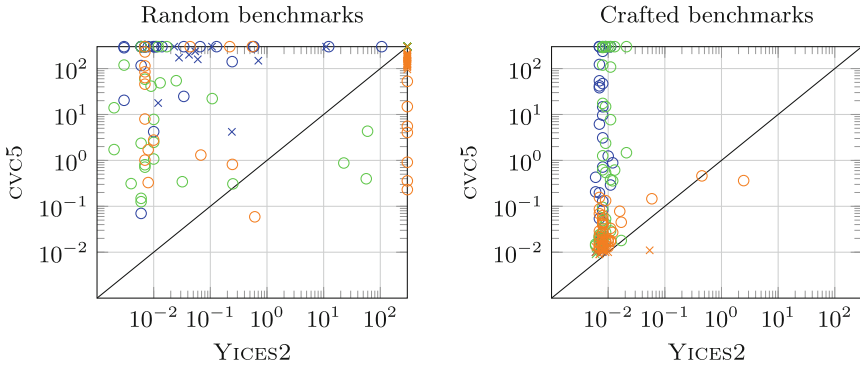


Fig. 2. Runtime comparison for benchmarks from [9] (in seconds, timeout 300 s) result: sat o, unsat x; finite field order: 3 (blue), 13 (green), and 211 (orange)

tends to be faster. We believe that this is due to the fact that the polynomials tend to be large (in the number of monomials), but rather easy to solve. Generating a full Gröbner basis upfront might add significant overhead.

For the second benchmark set, many instances can be solved by both solvers immediately (c.f. Fig. 3 right). We believe that those instances can be solved without extensive finite field reasoning, as the benchmark set contains Boolean structure. This enables both solvers to successfully solve benchmarks even with vast field orders. However, once extensive algebraic reasoning is required in finite fields of vast order (the majority of the benchmarks), the purely symbolic approach of CVC5 in proving unsatisfiability seems to be advantageous. An MCSat approach requires to pick actual values in a gigantic search space, thus especially strong lemmas need to be learned in order to prune the search space efficiently. Improving the explanation procedure is part of our future work.

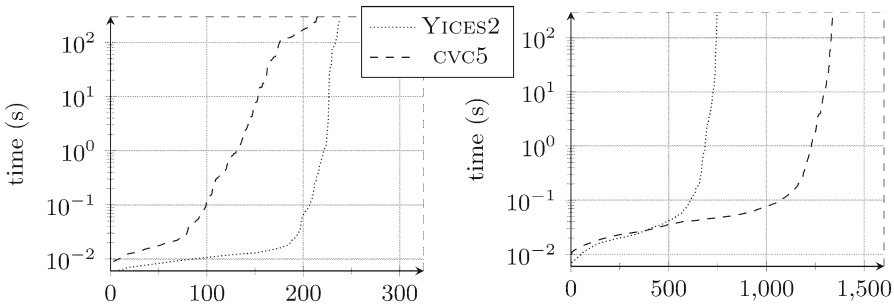


Fig. 3. Instances solved over time (timeout 300s) by YICES2 and CVC5 from [9] (left) and [17] (right).

6 Summary and Outlook

In this system description we have presented the first implementation of an MCSat-based decision procedure for non-linear finite field polynomials. We have shown that MCSat is a feasible way of solving SMT instances over finite fields and it compares well with SMT approaches using Gröbner bases for many instances.

The presented tool implementation is well suited for future experiments and the rapid development of more advanced explanation procedures that will eliminate the current bottlenecks with regard to large finite fields.

Acknowledgments. This work was conducted during the first author’s stay at SRI International. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the FWF grants SFB 10.55776/F8504 and ESPRIT 10.55776/ESP666), the WWTF ICT22-007 project ForSmart, the NSF award CCRI-2016597, the Amazon Research Award 2024 QuAT, and from SRI Internal Research And Development funds. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US Government or NSF.

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

References

1. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
3. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). <https://doi.org/10.3233/FAIA201017>, <https://doi.org/10.3233/FAIA201017>
4. Buchberger, B.: Bruno Buchberger’s PhD Thesis 1965: an algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *J. Symbol. Comput.* **41**(3-4), 475–511 (2006). <https://doi.org/10.1016/J.JSC.2005.09.007>
5. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
6. Dutertre, B., Goel, A., Graham-Lengrand, S., Irfan, A., Jovanovic, D., Mason, I.A.: Yices 2 in SMT-COMP 2023 (2023)
7. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. *SIAM J. Comput.* **18**(1), 186–208 (1989). <https://doi.org/10.1137/0218012>

8. Graham-Lengrand, S., Jovanović, D., Dutertre, B.: Solving Bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 103–121. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_7
9. Hader, T., Kaufmann, D., Kovács, L.: SMT solving over finite field arithmetic. In: Piskac, R., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR). EPIc Series in Computing, vol. 94, pp. 238–256. EasyChair (2023). <https://doi.org/10.29007/4N6W>, <https://doi.org/10.29007/4n6w>
10. Jovanović, D.: Solving nonlinear integer arithmetic with MCSAT. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 330–346. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_18
11. Jovanovic, D., Barrett, C., de Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 173–180. IEEE (2013). <https://doi.org/10.1109/FMCAD.2013.7027033>
12. Jovanovic, D., Dutertre, B.: Libpoly: a library for reasoning about polynomials. In: Brain, M., Hadarean, L. (eds.) Intl. Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings, vol. 1889, pp. 28–39. CEUR-WS.org (2017). <https://ceur-ws.org/Vol-1889/paper3.pdf>
13. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
14. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_1
15. de Moura, L.M., Bjørner, N.S.: Satisfiability modulo theories: introduction and applications. *Commun. ACM* **54**(9), 69–77 (2011)
16. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_3
17. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.W.: Satisfiability modulo finite fields. In: International Conference on Computer Aided Verification (CAV), Part II. LNCS, vol. 13965, pp. 163–186. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_8
18. Ozdemir, A., Wahby, R.S., Brown, F., Barrett, C.W.: Bounded verification for finite-field-blasting - in a compiler for zero knowledge proofs. In: Enea, C., Lal, A. (eds.) International Conference on Computer Aided Verification (CAV), Part III. LNCS, vol. 13966, pp. 154–175. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_8
19. Szabo, N.: Smart Contracts: Building Blocks for Digital Markets (1996). <http://www.fon.hum.uva.nl>
20. Wang, D.: Elimination Methods. Springer Science & Business Media (2001)









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

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





Certified MaxSAT Preprocessing

Hannes Ihalainen¹, Andy Oertel^{2,3}, Yong Kiam Tan⁴,
Jeremias Berg¹, Matti Järvisalo¹, Magnus O. Myreen⁵,
and Jakob Nordström^{2,3}

¹ Department of Computer Science, University of Helsinki, Helsinki, Finland
`{hannes.ihalainen, jeremias.berg, matti.jarvisalo}@helsinki.fi`

² Lund University, Lund, Sweden

`andy.oertel@cs.lth.se, jn@di.ku.dk`

³ University of Copenhagen, Copenhagen, Denmark

⁴ Institute for Infocomm Research (I2R), A*STAR, Singapore, Singapore
`tanyk1@i2r.a-star.edu.sg`

⁵ Chalmers University of Technology, Gothenburg, Sweden
`myreen@chalmers.se`

Abstract. Building on the progress in Boolean satisfiability (SAT) solving over the last decades, maximum satisfiability (MaxSAT) has become a viable approach for solving NP-hard optimization problems. However, ensuring correctness of MaxSAT solvers has remained a considerable concern. For SAT, this is largely a solved problem thanks to the use of proof logging, meaning that solvers emit machine-verifiable proofs to certify correctness. However, for MaxSAT, proof logging solvers have started being developed only very recently. Moreover, these nascent efforts have only targeted the core solving process, ignoring the preprocessing phase where input problem instances can be substantially reformulated before being passed on to the solver proper.

In this work, we demonstrate how pseudo-Boolean proof logging can be used to certify the correctness of a wide range of modern MaxSAT preprocessing techniques. By combining and extending the VERIPB and CAKEPB tools, we provide formally verified end-to-end proof checking that the input and preprocessed output MaxSAT problem instances have the same optimal value. An extensive evaluation on applied MaxSAT benchmarks shows that our approach is feasible in practice.

Keywords: maximum satisfiability · preprocessing · proof logging · formally verified proof checking

1 Introduction

The development of Boolean satisfiability (SAT) solvers is arguably one of the true success stories of modern computer science—today, SAT solvers are routinely used as core engines in many types of complex automated reasoning systems. One example of this is SAT-based optimization, usually referred to as

maximum satisfiability (MaxSAT) solving. The improved performance of SAT solvers, coupled with increasingly sophisticated techniques for using SAT solver calls to reason about optimization problems, have made MaxSAT solvers a powerful tool for tackling real-world NP-hard optimization problems [8].

However, Modern MaxSAT solvers are quite intricate pieces of software, and it has been shown repeatedly in the MaxSAT evaluations [51] that even the best solvers sometimes report incorrect results. This was previously a serious issue also for SAT solvers (see, e.g., [13]), but the SAT community has essentially eliminated this problem by requiring that solvers should be *certifying* [1, 53], i.e., not only report whether a given formula is satisfiable or unsatisfiable but also produce a machine-verifiable proof that this conclusion is correct. Many different SAT proof formats such as RUP [33], TRACECHECK [7], GRIT [17], and LRAT [16] have been proposed, with DRAT [35, 36, 74] established as the de facto standard; for the last ten years, proof logging has been compulsory in the (main track of the) SAT competitions [66]. It is all the more striking, then, that until recently no similar developments have been observed in MaxSAT solving.

1.1 Previous Work

A first natural question to ask—since MaxSAT solvers are based on repeated calls to SAT solvers—is why we cannot simply use SAT proof logging also for MaxSAT. The problem is that DRAT can only reason about clauses, whereas MaxSAT solvers argue about costs of solutions and values of objective functions. Translating such claims to clausal form would require an external tool to certify correctness of the translation. Also, such clausal translations incur a significant overhead and do not seem well-adapted for, e.g., counting arguments in MaxSAT.

While there have been several attempts to design proof systems specifically for MaxSAT solving [11, 23, 39, 45, 57, 58, 63–65], none of these have come close to providing a general proof logging solution, because they apply only for very specific algorithm implementations and/or fail to capture the full range of techniques used. Recent papers have instead proposed using pseudo-Boolean proof logging with VERIPB [9, 32] to certify correctness of so-called solution-improving solvers [72] and core-guided solvers [4]. Although these works demonstrate, for the first time, practical proof logging for modern MaxSAT solving, the methods developed thus far only apply to the core solving process. This ignores the preprocessing phase, where the input formula can undergo major reformulation. State-of-the-art solvers sometimes use stand-alone preprocessor tools, or sometimes integrate preprocessing-style reasoning more tightly within the MaxSAT solver engine, to speed up the search for optimal solutions. Some of these preprocessing techniques are lifted from SAT to MaxSAT, but there are also native MaxSAT preprocessing methods that lack analogies in SAT solving.

1.2 Our Contribution

In this paper, we show, for the first time, how to use pseudo-Boolean proof logging with VERIPB to produce proofs of correctness for a wide range of prepro-

cessing techniques used in modern MaxSAT solvers. VERIPB proof logging has previously been successfully used not only for core MaxSAT search as discussed above, but also for advanced SAT solving techniques (including symmetry breaking) [9, 27, 32], subgraph solving [28–30], constraint programming [22, 31, 54, 55], and 0–1 ILP presolving [37], and we add MaxSAT preprocessing to this list.

In order to do so, we extend the VERIPB proof format to include an *output section* where a reformulated output can be presented, and where the pseudo-Boolean proof establishes that this output formula and the input formula are *equioptimal*, i.e., have optimal solutions of the same value. We also enhance CAKEPB [10, 29]—a verified proof checker for pseudo-Boolean proofs—to handle proofs of reformulation. In this way, we obtain an end-to-end formally verified toolchain for certified preprocessing of MaxSAT instances.

It is worth noting that although preprocessing is also a critical component in SAT solving, we are not aware of any tool for certifying reformulations even for the restricted case of decision problems, i.e., showing that formulas are *equisatisfiable*—the DRAT format and tools support proofs that satisfiability of an input CNF formula F implies satisfiability of an output CNF formula G but not the converse direction (except in the special case where F is a subset of G). To the best of our knowledge, our work presents the first practical tool for proving (two-way) equisatisfiability or equioptimality of reformulated problems.

We have performed computational experiments running a MaxSAT preprocessor with proof logging and proof checking on benchmarks from the MaxSAT evaluations [51]. Although there is certainly room for improvements in performance, these experiments provide empirical evidence for the feasibility of certified preprocessing for real-world MaxSAT benchmarks.

1.3 Organization of This Paper

After reviewing preliminaries in Sect. 2, we explain our pseudo-Boolean proof logging for MaxSAT preprocessing in Sect. 3, and Sect. 4 discusses verified proof checking. We present results from a computational evaluation in Sect. 5, after which we conclude with a summary and outlook for future work in Sect. 6.

2 Preliminaries

We write ℓ to denote a literal, i.e., a $\{0, 1\}$ -valued Boolean variable x or its negation $\bar{x} = 1 - x$. A *clause* $C = \ell_1 \vee \dots \vee \ell_k$ is a disjunction of literals, where a *unit clause* consists of only one literal. A formula in *conjunctive normal form (CNF)* $F = C_1 \wedge \dots \wedge C_m$ is a conjunction of clauses, where we think of clauses and formulas as sets so that there are no repetitions and order is irrelevant.

A *pseudo-Boolean (PB) constraint* is a 0–1 linear inequality $\sum_j a_j \ell_j \geq b$, where, when convenient, we can assume all literals ℓ_j to refer to distinct variables and all integers a_j and b to be positive (so-called *normalized form*). A *pseudo-Boolean formula* is a conjunction of such constraints. We identify the clause $C =$

$\ell_1 \vee \dots \vee \ell_k$ with the pseudo-Boolean constraint $\text{PB}(C) = \ell_1 + \dots + \ell_k \geq 1$, so a CNF formula F is just a special type of PB formula $\text{PB}(F) = \{\text{PB}(C) \mid C \in F\}$.

A (*partial*) *assignment* ρ mapping variables to $\{0, 1\}$, extended to literals by respecting the meaning of negation, satisfies a PB constraint $\sum_j a_j \ell_j \geq b$ if $\sum_{\ell_j: \rho(\ell_j)=1} a_j \geq b$ (assuming normalized form). A PB formula is satisfied by ρ if all constraints in it are. We also refer to total satisfying assignments ρ as *solutions*. In a *pseudo-Boolean optimization (PBO)* problem we ask for a solution minimizing a given *objective function* $O = \sum_j c_j \ell_j + W$, where c_j and W are integers and W represents a trivial lower bound on the minimum cost.

2.1 Pseudo-Boolean Proof Logging Using Cutting Planes

The pseudo-Boolean proof logging in VERIPB is based on the *cutting planes* proof system [15] with extensions as discussed briefly next. We refer the reader to [14] for and in-depth discussion of cutting planes and to [9, 26, 37, 73] for more detailed information about the VERIPB proof system and format.

A pseudo-Boolean proof maintains two sets of *core constraints* \mathcal{C} and *derived constraints* \mathcal{D} under which the objective O should be minimized. At the start of the proof, \mathcal{C} is initialized to the constraints in the input formula F . Any constraints derived by the rules described below are placed in \mathcal{D} , from where they can later be moved to \mathcal{C} (but not vice versa). The proof system semantics preserves the invariant that the optimal value of any solution to \mathcal{C} and to the original input problem F is the same. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ by performing *addition* of two constraints or *multiplication* of a constraint by a positive integer, and *literal axioms* $\ell \geq 0$ can be used at any time. Additionally, we can apply *division* to $\sum_j a_j \ell_j \geq b$ by a positive integer d followed by rounding up to obtain $\sum_j \lceil a_j/d \rceil \ell_j \geq \lceil b/d \rceil$, and *saturation* to yield $\sum_j \min\{a_j, b\} \cdot \ell_j \geq b$ (where we again assume normalized form).

The negation of a constraint $C = \sum_j a_j \ell_j \geq b$ is $\neg C = \sum_j a_j \ell_j \leq b - 1$. For a (partial) assignment ρ we write $C|_\rho$ for the *restricted constraint* obtained by replacing literals in C assigned by ρ with their values and simplifying. We say that C *unit propagates* ℓ *under* ρ if $C|_\rho$ cannot be satisfied unless ℓ is assigned to 1. If repeated unit propagation on all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{-C\}$, starting with the empty assignment $\rho = \emptyset$, leads to contradiction in the form of an unsatisfiable constraint, we say that C follows by *reverse unit propagation (RUP)* from $\mathcal{C} \cup \mathcal{D}$. Such (efficiently verifiable) RUP steps are allowed in VERIPB proofs as a convenient way to avoid writing out an explicit cutting planes derivation. We use the same notation $C|_\omega$ to denote the result of applying to C a (*partial*) *substitution* ω , which can map variables not only to $\{0, 1\}$ but also to literals, and extend this notation to sets of constraints by taking unions.

In addition to the above rules, which derive semantically implied constraints, there is a *redundance-based strengthening rule*, or just *redundance rule* for short, that can derive non-implied constraints C as long as they do not change the feasibility or optimal value. This can be guaranteed by exhibiting a *witness substitution* ω such that for any total assignment α satisfying $\mathcal{C} \cup \mathcal{D}$ but violating C , the composition $\alpha \circ \omega$ is another total assignment that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and

yields an objective value that is at least as good. Formally, C can be derived from $\mathcal{C} \cup \mathcal{D}$ by exhibiting ω and subproofs for

$$\mathcal{C} \cup \mathcal{D} \cup \{-C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\}) \upharpoonright_{\omega} \cup \{O \geq O \upharpoonright_{\omega}\}, \tag{1}$$

using the previously discussed rules (where the notation $\mathcal{C}_1 \vdash \mathcal{C}_2$ means that the constraints \mathcal{C}_2 can be derived from the constraints \mathcal{C}_1).

During preprocessing, constraints in the input formula are often deleted or replaced by other constraints, in which case the proof should establish that these deletions maintain equioptimality. Removing constraints from the derived set \mathcal{D} is unproblematic, but unrestricted deletion from the core set \mathcal{C} can clearly introduce spurious better solutions. Therefore, removing C from \mathcal{C} can only be done by the *checked deletion rule*, which requires a proof that the redundancy rule can be used to rederive C from $\mathcal{C} \setminus \{C\}$ (see [9] for a more detailed explanation).

Finally, it turns out to be useful to allow replacing O by a new objective O' using an *objective function update rule*, as long as this does not change the optimal value of the problem. Formally, updating the objective from O to O' requires derivations of the two constraints $O \geq O'$ and $O' \geq O$ from the core set \mathcal{C} , which shows that any satisfying solution to \mathcal{C} has the same value for both objectives. More details on this rule can be found in [37].

2.2 Maximum Satisfiability

A WCNF instance of (weighted partial) maximum satisfiability $\mathcal{F}^W = (F_H, F_S)$ is a conjunction of two CNF formulas F_H and F_S with *hard* and *soft* clauses, respectively, where soft clauses $C \in F_S$ have positive weights w^C . A solution ρ to \mathcal{F}^W must satisfy F_H and has value $\text{COST}(F_S, \rho)$ equal to the sum of weights of all soft clauses not satisfied by ρ . The optimum $\text{OPT}(\mathcal{F}^W)$ of \mathcal{F}^W is the minimum of $\text{COST}(F_S, \rho)$ over all solutions ρ , or ∞ if no solution exists.

State-of-the-art MaxSAT preprocessors such as MAXPRE [39,44] take a slightly different *objective-centric* view [5] of MaxSAT instances $\mathcal{F} = (F, O)$ as consisting of a CNF formula F and an objective function $O = \sum_j c_j \ell_j + W$ to be minimized under assignments ρ satisfying F . A WCNF MaxSAT instance $\mathcal{F}^W = (F_H, F_S)$ is converted into objective-centric form $\text{OBJMAXSAT}(\mathcal{F}^W) = (F, O)$ by letting the formula $F = F_H \cup \{C \vee b_C \mid C \in F_S, |C| > 1\}$ of $\text{OBJMAXSAT}(\mathcal{F}^W)$ consist of the hard clauses of \mathcal{F}^W and the non-unit soft clauses in F_S , each extended with a fresh variable b_C that does not appear in any other clause. The objective $O = \sum_{(\bar{\ell}) \in F_S} w^{(\bar{\ell})} \ell + \sum w^C b_C$ contains literals ℓ for all unit soft clauses $\bar{\ell}$ in F_S as well as literals for all new variables b_C , with coefficients equal to the weights of the corresponding soft clauses. In other words, each unit soft clause $\bar{\ell} \in F_S$ of weight w is transformed into the term $w \cdot \ell$ in the objective function O , and each non-unit soft clause C is transformed into the hard clause $C \vee b_C$ paired with the unit soft clause (\bar{b}_C) with same weight as C . The following observation summarizes the properties of $\text{OBJMAXSAT}(\mathcal{F}^W)$ that are central to our work.

Observation 1. *For any solution ρ to a WCNF MaxSAT instance \mathcal{F}^W there exists a solution ρ' to $(F, O) = \text{OBJMAXSAT}(\mathcal{F}^W)$ with $O(\rho') = \text{COST}(\mathcal{F}^W, \rho)$. Conversely, if ρ' is a solution to $\text{OBJMAXSAT}(\mathcal{F}^W)$, then there exists a solution ρ of \mathcal{F}^W for which $\text{COST}(\mathcal{F}^W, \rho) \leq O(\rho')$.*

For the second part of the observation, the reason $O(\rho')$ is only an upper bound on $\text{COST}(\mathcal{F}^W, \rho)$ is that the encoding forces b_C to be true whenever C is not satisfied by an assignment but not vice versa.

An objective-centric MaxSAT instance (F, O) , in turn, clearly has the same optimum as the pseudo-Boolean optimization problem of minimizing O subject to $\text{PB}(F)$. For the end-to-end formal verification, the fact that this coincides with $\text{OPT}(\mathcal{F}^W)$ needs to be formalized into theorems as shown in Fig. 4.

3 Proof Logging for MaxSAT Preprocessing

We now discuss how pseudo-Boolean proof logging can be used to reason about correctness of MaxSAT preprocessing steps. Our approach maintains the invariant that the current working instance in the preprocessor is synchronized with the PB constraints in the core set \mathcal{C} as described in Sect. 2.2. At the end of each preprocessing step (i.e., application of a preprocessing technique) the set of derived constraints \mathcal{D} is empty. All constraints derived in the proof as described in this section are moved to the core set, and constraints are always removed by checked deletion from the core set. Full details are in the online appendix [40].

3.1 Overview

All our preprocessing steps maintain *equioptimality*, which means that if preprocessing of the WCNF MaxSAT instance \mathcal{F}^W yields the output instance \mathcal{F}_P^W , then the equality $\text{OPT}(\mathcal{F}^W) = \text{OPT}(\mathcal{F}_P^W)$ is guaranteed to hold. Our preprocessing is *certified*, meaning that we provide a machine-verifiable proof justifying this claimed equality. Our discussion below focuses on input instances that have solutions, but our techniques also handle the—arguably less interesting—case of \mathcal{F}^W not having solutions; details are in the online appendix [40].

An overview of the workflow of our certifying MaxSAT preprocessor is shown in Fig. 1. Given a WCNF instance \mathcal{F}^W as input, the preprocessor proceeds in five stages (illustrated on the left in Fig. 1), and then outputs a preprocessed MaxSAT instance \mathcal{F}_P^W together with a pseudo-Boolean proof that $\text{OPT}(\text{OBJMAXSAT}(\mathcal{F}^W)) = \text{OPT}(\text{OBJMAXSAT}(\mathcal{F}_P^W))$. For certified MaxSAT preprocessing, this proof can then be fed to a formally verified checker as in Sect. 4 to verify that (a) the initial core constraints in the proof correspond exactly to the clauses in $\text{OBJMAXSAT}(\mathcal{F}^W)$, (b) each step in the proof is valid, and (c) the final core constraints in the proof correspond exactly to the clauses in $\text{OBJMAXSAT}(\mathcal{F}_P^W)$. Below, we provide more details on the five stages of the preprocessing flow.

	<i>preprocessing</i> (<i>MaxSAT</i>)	<i>proof</i> (<i>pseudo-Boolean</i>)
1. Initialization	$(\mathcal{F}^W, 0)$	$(\text{PB}(F^0), O^0)$ where $(F^0, O^0) = \text{ObjMaxSAT}(\mathcal{F}^W)$
2. Preprocessing on WCNF	$(\mathcal{F}_1^W, \text{LB}^1)$	(\mathcal{C}^1, O^1)
3. Conversion to objective-centric	$(F^2, O^2 + \text{LB}^1)$ where $(F^2, O^2) = \text{ObjMaxSAT}(\mathcal{F}_1^W)$	$(\text{PB}(F^2), O^2 + \text{LB}^1)$
4. Preprocessing on objective-centric	(F^3, O^3)	$(\text{PB}(F^3), O^3)$
5. Constant removal	(F^4, O^4) where $F^4 = F^3 \wedge (b^{W^3})$ $O^4 = O^3 - W^3 + W^3 b^{W^3}$	$(\text{PB}(F^4), O^4)$
Output	Preprocessed WCNF $\mathcal{F}_P^W = (F^4, F_S^P)$	Proof of equioptimality of $\text{PB}(F^0)$ under O^0 and $\text{PB}(F^4)$ under O^4

Fig. 1. Overview of the five stages of certified MaxSAT preprocessing of a WCNF instance \mathcal{F}^W . The middle column contains the state of the working MaxSAT instance as a WCNF instance and a lower bound on its optimum cost (Stages 1–2), or as an objective-centric instance (Stages 3–5). The right column contains a tuple (\mathcal{C}, O) with the set \mathcal{C} of core constraints, and objective O , respectively, of the proof after each stage.

Stage 1: Initialization. An input WCNF instance \mathcal{F}^W is transformed to pseudo-Boolean format by converting it to an objective-centric representation $(F^0, O^0) = \text{ObjMaxSAT}(\mathcal{F}^W)$ and then representing all clauses in F^0 as pseudo-Boolean constraints as described in Sect. 2.2. The VERIPB proof starts out with core constraints $\text{PB}(F^0)$ and objective O^0 . The preprocessor maintains a lower bound on the optimal cost of the working instance, which is initialized to 0 for the input \mathcal{F}^W .

Stage 2: Preprocessing on the Initial WCNF Representation. During preprocessing on the WCNF representation, a (very limited) set of simplification techniques are applied on the working formula. At this stage the preprocessor removes duplicate, tautological, and blocked clauses [43]. Additionally, hard unit clauses are unit propagated and clauses subsumed by hard clauses are removed. Importantly, the preprocessor is performing these simplifications on a WCNF MaxSAT instance where it deals with hard and soft clauses. As the pseudo-Boolean proof has no concept of hard or soft clauses, the reformulation steps must be expressed in terms of the constraints in the proof. The next example illustrates how reasoning with different types of clauses is logged in the proof.

Example 1. Suppose the working instance has two duplicate clauses C and D . If both are hard, then the proof has two identical constraints $\text{PB}(C)$ and $\text{PB}(D)$ in the core set, and $\text{PB}(D)$ can be deleted since it follows from $\text{PB}(C)$ by reverse unit propagation (RUP). If D is instead a non-unit soft clause, the proof has the constraint $\text{PB}(D \vee b_D)$ and the term $w^D b_D$ in the objective, where b_D does not appear in any other constraint. Then in the proof we (1) remove the RUP constraint $\text{PB}(D \vee b_D)$, (2) introduce $\bar{b}_D \geq 1$ by redundancy-based strengthening using the witness $\{b_D \rightarrow 0\}$, (3) remove the term $w^D b_D$ from the objective, and (4) delete $\bar{b}_D \geq 1$ with the witness $\{b_D \rightarrow 0\}$.

Stage 3: Conversion to Objective-Centric Representation. In order to apply more simplification rules in a cost-preserving way, the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ at the end of Stage 2 is converted into the corresponding objective-centric representation that takes the lower-bound LB inferred during Stage 1 into account. More specifically, the preprocessor next converts its working MaxSAT instance into the objective-centric instance $\mathcal{F}_2 = (F^2, O^2 + \text{LB})$ where $(F^2, O^2) = \text{OBJMAXSAT}(\mathcal{F}_1^W)$.

Here it is important to note that at the end of Stage 2, the core constraints \mathcal{C}^1 and objective O^1 of the proof are not necessarily $\text{PB}(F^2)$ and $O^2 + \text{LB}$, respectively. Specifically, consider a unit soft clause $(\bar{\ell})$ of \mathcal{F}_1^W obtained by shrinking a non-unit soft clause $C \supseteq (\bar{\ell})$ of the input instance, with weight w^C . Then the objective function O^2 in the preprocessor will include the term $w^C \ell$ that does not appear in the objective function O^1 in the proof. Instead, O^1 contains the term $w^C b_C$ and \mathcal{C}^1 the constraint $\bar{\ell} + b_C \geq 1$ where b_C is the fresh variable added to C in Stage 1. In order to “sync up” the working instance and the proof we (1) introduce $\ell + \bar{b}_C \geq 1$ to the proof with the witness $\{b_C \rightarrow 0\}$, (2) update O^1 by adding $w^C \ell - w^C b_C$, (3) remove the constraint $\ell + \bar{b}_C \geq 1$ with the witness $\{b_C \rightarrow 0\}$, and (4) remove the constraint $\bar{\ell} + b_C \geq 1$ with witness $\{b_C \rightarrow 1\}$. The same steps are logged for all soft unit clauses of \mathcal{F}_1^W obtained during Stage 2. In the following stages, the preprocessor will operate on an objective-centric MaxSAT instance whose clauses correspond exactly to the core constraints of the proof.

Stage 4: Preprocessing on the Objective-Centric Representation. During preprocessing on the objective-centric representation, more simplification techniques are applied to the working objective-centric instance and logged to the proof. We implemented proof logging for a wide range of preprocessing techniques. These include MaxSAT versions of rules commonly used in SAT solving like bounded variable elimination (BVE) [20, 68], bounded variable addition [49], blocked clause elimination [43], subsumption elimination, self-subsuming resolution [20, 60], failed literal elimination [24, 46, 75], and equivalent literal substitution [12, 48, 71]. We also cover MaxSAT-specific preprocessing rules like TrimMaxSAT [61], (group)-subsumed literal (or label) elimination (SLE) [6, 44], intrinsic at-most-ones [38, 39], binary core removal (BCR) [25, 44], label matching [44], and hardening [2, 39, 56]. Here we give examples for BVE, SLE, label

matching, and BCR—the rest are detailed in the online appendix [40]. In the following descriptions, let (F, O) be the current objective-centric working instance.

Bounded Variable Elimination (BVE) [20, 68]. BVE eliminates from F a variable x that does not appear in the objective by replacing all clauses in which either x or \bar{x} appears with the non-tautological clauses in $\{C \vee D \mid C \vee x \in F, D \vee \bar{x} \in F\}$.

An application of BVE is logged as follows: (1) each non-tautological constraint $\text{PB}(C \vee D)$ is added by summing the existing constraints $\text{PB}(C \vee x)$ and $\text{PB}(D \vee \bar{x})$ and saturating, after which (2) each constraint of the form $\text{PB}(C \vee x)$ and $\text{PB}(D \vee \bar{x})$ is deleted with the witness $x \rightarrow 1$ or $x \rightarrow 0$, respectively.

Label Matching [44]. Label matching allows merging pairs of objective variables that can be deduced to not both be set to 1 by optimal solutions. Assume that (i) F contains the clauses $C \vee b_C$ and $D \vee b_D$, (ii) b_C and b_D are objective variables with the same coefficient w in O , and (iii) $C \vee D$ is a tautology. Then label matching replaces b_C and b_D with a fresh variable b_{CD} , i.e., replaces $C \vee b_C$ and $D \vee b_D$ with $C \vee b_{CD}$ and $D \vee b_{CD}$ and adds $-wb_C - wb_D + wb_{CD}$ to O .

As $C \vee D$ is a tautology there is some literal ℓ such that $\bar{\ell} \in C$ and $\ell \in D$. Label matching is logged via the following steps: (1) introduce the constraint $\bar{b}_C + \bar{b}_D \geq 1$ with the witness $\{b_C \rightarrow \ell, b_D \rightarrow \bar{\ell}\}$, (2) introduce the constraints $b_{CD} + \bar{b}_C + \bar{b}_D \geq 2$ and $\bar{b}_{CD} + b_C + b_D \geq 1$ by redundancy; these correspond to $b_{CD} = b_C + b_D$ which holds even though the variables are binary due to the constraint added in the first step, (3) update the objective by adding $-wb_C - wb_D + wb_{CD}$ to it, (4) introduce the constraints $\text{PB}(C \vee b_{CD})$ and $\text{PB}(D \vee b_{CD})$ which are RUP, (5) delete $\text{PB}(C \vee b_C)$ and $\text{PB}(D \vee b_D)$ with the witness $\{b_C \rightarrow \bar{\ell}, b_D \rightarrow \ell\}$, (6) delete the constraint $b_{CD} + \bar{b}_C + \bar{b}_D \geq 2$ with the witness $\{b_C \rightarrow 0, b_D \rightarrow 0\}$ and $\bar{b}_{CD} + b_C + b_D \geq 1$ with the witness $\{b_C \rightarrow 1, b_D \rightarrow 0\}$, (7) delete $\bar{b}_C + \bar{b}_D \geq 1$ with the witness $\{b_C \rightarrow 0\}$.

Subsumed Literal Elimination (SLE) [6, 39]. Given two non-objective variables x and y such that (i) $\{C \mid C \in F, y \in C\} \subseteq \{C \mid C \in F, x \in C\}$ and (ii) $\{C \mid C \in F, \bar{x} \in C\} \subseteq \{C \mid C \in F, \bar{y} \in C\}$, subsumed literal elimination (SLE) allows fixing $x = 1$ and $y = 0$. This is proven by (1) introducing $x \geq 1$ and $\bar{y} \geq 1$, both with witness $\{x \rightarrow 1, y \rightarrow 0\}$, (2) simplifying the constraint database via propagation, and (3) deleting the constraints introduced in the first step as neither x nor y appears in any other constraints after simplification.

If x and y are objective variables, the application of SLE additionally requires that: (iii) the coefficient in the objective of x is at most as high as the coefficient of y . Then the value of x is not fixed as it would incur cost. Instead, only $y = 0$ is fixed and y removed from the objective. Intuitively, conditions (i) and (ii) establish that the values of x and y can always be flipped to 0 and 1, respectively, without falsifying any clauses. If neither of the variables is in the objective, this flip does not increase the cost of any solutions. Otherwise, condition (iii) ensures that the flip does not make the solution worse, i.e., increase its cost.

Binary Core Removal (BCR) [25,44]. Assume that the following four prerequisites hold: (i) F contains a clause $b_C \vee b_D$ for two objective variables b_C and b_D , (ii) b_C and b_D have the same coefficient w in O , (iii) the negations \bar{b}_C and \bar{b}_D do not appear in any clause of F , and (iv) both b_C and b_D appear in at least one other clause of F but not together in any other clause of F . Binary core removal replaces all clauses containing b_C or b_D with the non-tautological clauses in $\{C \vee D \vee b_{CD} \mid C \vee b_C \in F, D \vee b_D \in F\}$, where b_{CD} is a fresh variable, and modifies the objective function by adding $-wb_C - wb_D + wb_{CD} + w$ to it.

BCR is logged as a combination of the so-called *intrinsic at-most-ones* technique [38,39] and BVE. Applying intrinsic at most ones on the variables b_C and b_D introduces a new clause $(\bar{b}_C \vee \bar{b}_D \vee b_{CD})$ and adds $-wb_C - wb_D + wb_{CD} + w$ to the objective. Our proof for intrinsic at most ones is the same as the one presented in [4]. As this step removes b_C and b_D from the objective, both can now be eliminated via BVE.

Stage 5: Constant Removal and Output. After objective-centric preprocessing, the final objective-centric instance (F^3, O^3) is converted back to a WCNF instance. Before doing so, the constant term W_3 of O^3 is removed by introducing a fresh variable b^{W_3} , and setting $F^4 = F^3 \wedge (b^{W_3})$ and $O^4 = O^3 - W_3 + W_3b^{W_3}$. This step is straightforward to prove.

Finally, the preprocessor outputs the WCNF instance $\mathcal{F}_P^W = (F^4, F_S^P)$ that has F^4 as hard clauses. The set F_S^P of soft clauses consists of a unit soft clause $(\bar{\ell})$ of weight c for each term $c \cdot \ell$ in O^4 . The preprocessor also outputs the final proof of the fact that the minimum-cost of solutions to the pseudo-Boolean formula $\text{PB}(F^0)$ under O^0 is the same as that of $\text{PB}(F^4)$ under O^4 , i.e. that $\text{OPT}(\text{OBJMAXSAT}(\mathcal{F}^W)) = \text{OPT}(\text{OBJMAXSAT}(\mathcal{F}_P^W))$.

3.2 Worked Example of Certified Preprocessing

We give a worked-out example of certified preprocessing of the instance $\mathcal{F}^W = (F_H, F_S)$ where $F_H = \{(x_1 \vee x_2), (\bar{x}_2)\}$ and three soft clauses: (\bar{x}_1) with weight 1, $(x_3 \vee \bar{x}_4)$ with weight 2, and $(x_4 \vee \bar{x}_5)$ with weight 3. The proof for one possible execution of the preprocessor on this input instance is detailed in Table 1.

During Stage 1 (Steps 1–4 in Table 1), the core constraints of the proof are initialized to contain the four constraints corresponding to the hard and non-unit soft clauses of \mathcal{F}^W (IDs (1)–(4) in Table 1), and the objective to $x_1 + 2b_1 + 3b_2$, where b_1 and b_2 are fresh variables added to the non-unit soft clauses of \mathcal{F}^W .

During Stage 2 (Steps 5–9), the preprocessor fixes $x_2 = 0$ via unit propagation by removing x_2 from the clause $(x_1 \vee x_2)$, and then removing the unit clause (\bar{x}_2) . The justification for fixing $x_2 = 0$ are Steps 5–7. Next the preprocessor fixes $x_1 = 1$ which (i) removes the hard clause (x_1) , and (ii) increases the lower bound on the optimal cost by 1. The justification for fixing $x_1 = 1$ are Steps 8 and 9 of Table 1. At this point—at the end of Stage 2—the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ has $F_H^1 = \{\}$ and $F_S^1 = \{(x_3 \vee \bar{x}_4), (x_4 \vee \bar{x}_5)\}$.

Table 1. Example proof produced by a certifying preprocessor. The column (ID) refers to constraint IDs in the pseudo-Boolean proof. The column (Step) indexes all proof logging steps and is used when referring to the steps in the discussion. The letter ω is used for the witness substitution in redundance-based strengthening steps.

Step	ID	Type	Justification	Objective
1	(1)	add $x_1 + x_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
2	(2)	add $\bar{x}_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
3	(3)	add $x_3 + \bar{x}_4 + b_1 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
4	(4)	add $x_4 + \bar{x}_5 + b_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
<i>Unit propagation: fix $x_2 = 0$, constraint (2)</i>				
5	(5)	add $x_1 \geq 1$	(1) + (2)	$x_1 + 2b_1 + 3b_2$
6		delete (1)	RUP	$x_1 + 2b_1 + 3b_2$
7		delete (2)	$\omega: \{x_2 \rightarrow 0\}$	$x_1 + 2b_1 + 3b_2$
<i>Unit propagation; fix $x_1 = 1$, constraint (5)</i>				
8		add $-x_1 + 1$ to obj	(5)	$2b_1 + 3b_2 + 1$
9		delete (5)	$\omega: \{x_1 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
<i>BVE: eliminate x_4</i>				
10	(6)	add $x_3 + b_1 + \bar{x}_5 + b_2 \geq 1$	(3) + (4)	$2b_1 + 3b_2 + 1$
11		delete (3)	$\omega: \{x_4 \rightarrow 0\}$	$2b_1 + 3b_2 + 1$
12		delete (4)	$\omega: \{x_4 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
<i>Subsumed literal elimination: \bar{b}_2</i>				
13	(7)	add $\bar{b}_2 \geq 1$	$\omega: \{b_2 \rightarrow 0, b_1 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
14		add $-3b_2$ to obj	(7)	$2b_1 + 1$
15	(8)	add $x_3 + b_1 + \bar{x}_5 \geq 1$	(6) + (7)	$2b_1 + 1$
16		delete (6)	RUP	$2b_1 + 1$
17		delete (7)	$\omega: \{b_2 \rightarrow 0\}$	$2b_1 + 1$
<i>Remove objective constant</i>				
18	(9)	add $b_3 \geq 1$	$\omega: \{b_3 \rightarrow 1\}$	$2b_1 + 1$
19		add $b_3 - 1$ to obj	(9)	$2b_1 + b_3$

In Stage 3, the preprocessor converts its working instance into the objective-centric representation (F, O) where $F = \{(x_3 \vee \bar{x}_4 \vee b_1), (x_4 \vee \bar{x}_5 \vee b_2)\}$ and $O = 2b_1 + 3b_2 + 1$, which exactly matches the core constraints and objective of the proof after Step 9. Thus, in this instance, the conversion does not result in any proof logging steps. Afterwards, during Stage 4 (Steps 10–17), the preprocessor applies BVE in order to eliminate x_4 (Steps 10–12) and SLE to fix b_2 to 0 (Steps 13–17). Finally, Steps 18 and 19 represent Stage 5, i.e., the removal of the constant 1 from the objective. After these steps, the preprocessor outputs the preprocessed instance $\mathcal{F}_P^W = (F_H^P, F_S^P)$, where $F_H^P = \{(x_3 \vee \bar{x}_5 \vee b_1), (b_3)\}$ and F_S^P contains two clauses: (\bar{b}_1) with weight 2, and (\bar{b}_3) with weight 1.

4 Verified Proof Checking for Preprocessing Proofs

This section presents our new workflow for formally verified, end-to-end proof checking of MaxSAT preprocessing proofs based on pseudo-Boolean reasoning; an overview of this workflow is shown in Fig. 2. To realize this workflow, we extended the VERIPB tool and its proof format to support a new *output section* for declaring (and checking) reformulation guarantees between input and output PBO instances (Sect. 4.1); we similarly modified CAKEPB [29] a verified proof checker to support the updated proof format (Sect. 4.2); finally, we built a verified frontend, CAKEPBWCNF, which mediates between MaxSAT WCNF instances and PBO instances (Sect. 4.3). Our formalization is carried out in the HOL4 proof assistant [67] using CAKEML tools [34, 59, 70] to obtain a verified executable implementation of CAKEPBWCNF.

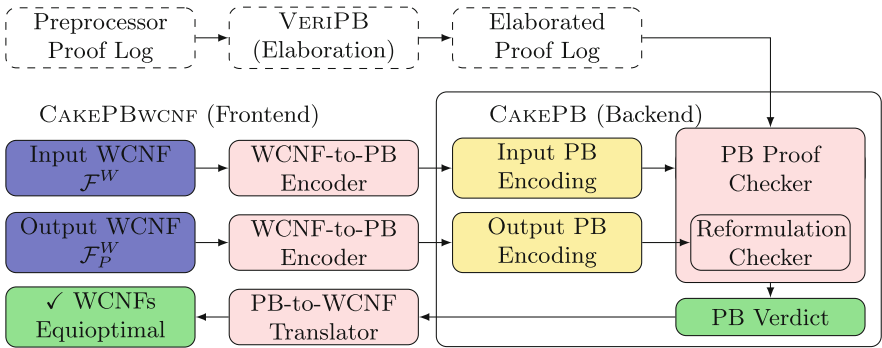


Fig. 2. Workflow for end-to-end verified MaxSAT preprocessing proof checking.

In the workflow in Fig. 2, the MaxSAT preprocessor produces a reformulated output WCNF together with a proof of equioptimality with the input WCNF. This proof is elaborated by VERIPB and then checked by CAKEPBWCNF, resulting in a verified *verdict*—in case of success, the input and output WCNFs are equioptimal. This workflow also supports verified checking of WCNF MaxSAT solving proofs (where the output parts of the flow are omitted).

4.1 Output Section for Pseudo-Boolean Proofs

Given an input PBO instance (F, O) , the VERIPB proof system as described in Sect. 2.1 maintains the invariant that the core constraints \mathcal{C} (and current objective) are equioptimal to the input instance. Utilizing this invariant, the new *output section* for VERIPB proofs allows users to optionally specify an output PBO instance (F', O') at the end of a proof. This output instance is claimed to be a reformulation of the input which is either: (i) *derivable*, i.e., satisfiability of F implies satisfiability of F' , (ii) *equisatisfiable* to F , or (iii)

equioptimal to (F, O) . These are increasingly stronger claims about the relationship between the input and output instances. After checking a pseudo-Boolean derivation, VERIPB runs reformulation checking which, e.g., for equioptimality, checks that $\mathcal{C} \subseteq F'$, $F' \subseteq \mathcal{C}$, and that the respective objective functions are syntactically equal after normalization; other reformulation guarantees are checked analogously.

The VERIPB tool supports an *elaboration* mode [29], where in addition to checking the proof it also converts it from *augmented format* to *kernel format*. The augmented format contains syntactic sugar rules to facilitate proof logging for solvers and preprocessors like MAXPRE, while the kernel format is supported by the formally verified proof checker CAKEPB. The new output section is passed unchanged from augmented to kernel format during elaboration.

4.2 Verified Proof Checking for Reformulations

There are two main verification tasks involved in extending CAKEPB with support for the output section. The first task is to verify soundness of all cases of reformulation checking. Formally, the equioptimality of an input PBO instance fml , obj and its output counterpart fml' , obj' is specified as follows:

$$\begin{aligned} \text{sem_output } fml \text{ } obj \text{ None } fml' \text{ } obj' \text{ Equioptimal} &\stackrel{\text{def}}{=} \\ \forall v. (\exists w. \text{satisfies } w \text{ } fml \wedge \text{eval_obj } obj \text{ } w \leq v) &\iff \\ (\exists w'. \text{satisfies } w' \text{ } fml' \wedge \text{eval_obj } obj' \text{ } w' \leq v) & \end{aligned}$$

This definition says that, for all values v , the input instance has a satisfying assignment with objective value less than or equal to v iff the output instance also has such an assignment; note that this implies (as a special case) that fml is satisfiable iff fml' is satisfiable. The verified correctness theorem for CAKEPB says that *if* CAKEPB successfully checks a pseudo-Boolean proof in kernel format and prints a verdict declaring equioptimality, then the input and output instances are indeed equioptimal as defined in `sem_output`.

The second task is to develop verified optimizations to speedup proof steps which occur frequently in preprocessing proofs; some code hotspots were also identified by profiling the proof checker against proofs generated by MAXPRE. Similar (unverified) versions of these optimizations are also used in VERIPB. These optimizations turned out to be necessary in practice—they mostly target steps which, when naïvely implemented, have quadratic (or worse) time complexity in the size of the constraint database.

Optimizing Reformulation Checking. The most expensive step in reformulation checking for the output section is to ensure that the core constraints \mathcal{C} are included in the output formula and vice versa (possibly with permutations and duplicity). Here, CAKEPB normalizes all pseudo-Boolean constraints involved to a canonical form and then copies both \mathcal{C} and the output formula into respective array-backed hash tables for fast membership tests.

Optimizing Redundance and Checked Deletion Rules. A naïve implementation of these two rules would require iterating over the entire constraints database when checking all subproofs in (1) for the right-hand-side constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})|_{\omega} \cup \{O \geq O|_{\omega}\}$. An important observation here is that preprocessing proofs frequently use substitutions ω that only involve a small number of variables (often a single variable, which in addition is fresh in the important special case of *reification* constraints $z \Leftrightarrow C$ encoding that z is true precisely when the constraint C is satisfied). Consequently, most of the constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})|_{\omega}$ can be skipped when checking redundance because they are unchanged by the substitution. Similarly, the constraint $O \geq O|_{\omega}$ is expensive to construct when the objective O contains many terms, but this construction can be skipped if no variables being substituted occur in O . CAKEPB stores a lazily-updated mapping of variables to their occurrences in the constraint database and the objective, which it uses to detect these cases.

The occurrence mapping just discussed is crucial for performance due to the frequency of steps involving witnesses for preprocessing proofs, but incurs some memory overhead in the checker. More precisely, every variable occurrence in any constraint in the database corresponds to exactly one ID in the mapping. Thus, the overhead of storing the mapping is in the worst case quadratic in the number of constraints, but it is still linear in the total space usage for the constraints database.

4.3 Verified WCNF Frontend

The CAKEPBWCNF frontend mediates between MaxSAT WCNF problems and pseudo-Boolean optimization problems native to CAKEPB. Accordingly, the correctness of CAKEPBWCNF is stated in terms of MaxSAT semantics, i.e., the encoding, underlying pseudo-Boolean semantics, and proof system are all formally verified. In order to trust CAKEPBWCNF, one *only* has to carefully inspect the formal definition of MaxSAT semantics shown in Fig. 3 to make sure that it matches the informal definition in Sect. 2.2. Here, each clause C is paired with a natural number n , where $n = 0$ indicates a hard clause and when $n > 0$ it is the weight of C . The optimal cost of a weighted CNF formula $wfml$ is `None` (representing ∞) if no satisfying assignment to the hard clauses exist; otherwise, it is the minimum cost among all satisfying assignments to the hard clauses.

$$\begin{aligned}
 \text{sat_hard } w \text{ } wfml &\stackrel{\text{def}}{=} \forall C. \text{mem } (0, C) \text{ } wfml \Rightarrow \text{sat_clause } w \ C \\
 \text{weight_clause } w \ (n, C) &\stackrel{\text{def}}{=} \text{if } \text{sat_clause } w \ C \text{ then } 0 \text{ else } n \\
 \text{cost } w \ wfml &\stackrel{\text{def}}{=} \text{sum } (\text{map } (\text{weight_clause } w) \ wfml) \\
 \text{opt_cost } wfml &\stackrel{\text{def}}{=} \text{if } \neg \exists w. \text{sat_hard } w \ wfml \text{ then } \text{None} \\
 &\quad \text{else } \text{Some } (\min_{\text{set}} \{ \text{cost } w \ wfml \mid \text{sat_hard } w \ wfml \})
 \end{aligned}$$

Fig. 3. Formalized semantics for MaxSAT WCNF problems.

There and Back Again. CAKEPBWCNF contains a verified WCNF-to-PB encoder implementing the encoding described in Sect. 2.2. Its correctness theorems are shown in Fig. 4, where the two lemmas in the top row relate the satisfiability and cost of the WCNF to its PB optimization counterpart after running `wcnf_to_pbf` (and vice versa), see Observation 1. Using these lemmas, the final theorem (bottom row) shows that equioptimality for two (encoded) PB optimization problems can be *translated* back to equioptimality for the input and preprocessed WCNFs.

$$\begin{array}{l}
\vdash \text{wfm1_to_pbf } wfml = (obj, pbf) \wedge \\
\text{satisfies } w \text{ (set } pbf) \Rightarrow \\
\exists w'. \text{sat_hard } w' wfml \wedge \\
\text{cost } w' wfml \leq \text{eval_obj } obj \ w
\end{array}
\qquad
\begin{array}{l}
\vdash \text{wfm1_to_pbf } wfml = (obj, pbf) \wedge \\
\text{sat_hard } w wfml \Rightarrow \\
\exists w'. \text{satisfies } w' \text{ (set } pbf) \wedge \\
\text{eval_obj } obj \ w' = \text{cost } w wfml
\end{array}$$

$$\begin{array}{l}
\vdash \text{full_encode } wfml = (obj, pbf) \wedge \text{full_encode } wfml' = (obj', pbf') \wedge \\
\text{sem_output (set } pbf) \text{ obj None (set } pbf') \text{ obj' Equioptimal} \Rightarrow \\
\text{opt_cost } wfml = \text{opt_cost } wfml'
\end{array}$$

Fig. 4. Correctness theorems for the WCNF-to-PB encoding.

Putting Everything Together. The final verification step is to specialize the end-to-end machine code correctness theorem for CAKEPB to the new frontend. The resulting theorem for CAKEPBWCNF is shown abridged in Fig. 5; a detailed explanation of similar CAKEML-based theorems is available elsewhere [29, 69] so we do not go into details here. Briefly, the theorem says that whenever the verdict string “s VERIFIED OUTPUT EQUIOPTIMAL” is printed (as a suffix) to the standard output by an execution of CAKEPBWCNF, then the two input files given on the command line parsed to equioptimal MaxSAT WCNF instances.

$$\begin{array}{l}
\vdash \text{cake_pb_wcnf_run } cl \ fs \ mc \ ms \Rightarrow \\
\exists \text{out err.} \\
\text{extract_fs } fs \ (\text{cake_pb_wcnf_io_events } cl \ fs) = \\
\text{Some (add_stdout (add_stderr } fs \ \text{err) } \text{out}) \wedge \\
(\text{length } cl = 4 \wedge \text{isSuffix "s VERIFIED OUTPUT EQUIOPTIMAL\n"} \ \text{out}) \Rightarrow \\
\exists wfml wfml'. \\
\text{get_fml } fs \ (\text{el } 1 \ cl) = \text{Some } wfml \wedge \text{get_fml } fs \ (\text{el } 3 \ cl) = \text{Some } wfml' \wedge \\
\text{opt_cost } wfml = \text{opt_cost } wfml'
\end{array}$$

Fig. 5. Abridged final correctness theorem for CAKEPBWCNF.

5 Experiments

We updated the MaxSAT preprocessor MAXPRE 2.1 [39, 42, 44] to MAXPRE 2.2 which now produces proof logs in the VERIPB format [10]. MAXPRE 2.2 is

available at the MAXPRE 2 repository [50]. The generated proofs were elaborated using VERIPB [73] and then checked by the verified proof checker CAKEPBWCNF. As benchmarks we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [52].

The experiments were conducted on 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPUs with 16 GB of memory, a solid state drive as storage, and Rocky Linux 8.5 as operating system. Each benchmark ran exclusively on a node and the memory was limited to 14 GB. The time for MAXPRE was limited to 300s. There is an option to let MAXPRE know about this time limit, but we did not use this option since MAXPRE then decides which techniques to try based on how much time remains. This would have made it very hard to get reliable measurements of the overhead when proof logging is switched on in the preprocessor. The time limits for both VERIPB and CAKEPBWCNF were set to 6000s to get as many instances checked as possible.

The main focus of our evaluation was the default setting of MAXPRE, which does not use some of the techniques mentioned in Sect. 3 (or the online appendix [40]). We also conducted experiments with all techniques enabled to check the correctness of the proof logging implementation for all preprocessing techniques. The data and source code from our experiments can be found in [41].

The goal of the experiments was to answer the following questions:

RQ1. How much extra time is required to write the proof for the preprocessor?

RQ2. How long does proof checking take compared to proof generation?

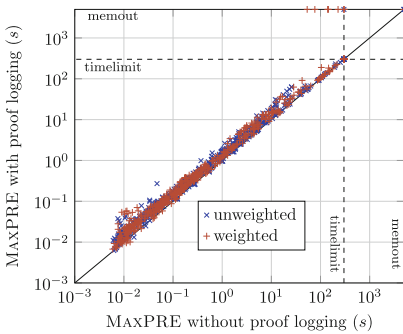


Fig. 6. Proof logging overhead for MAXPRE.

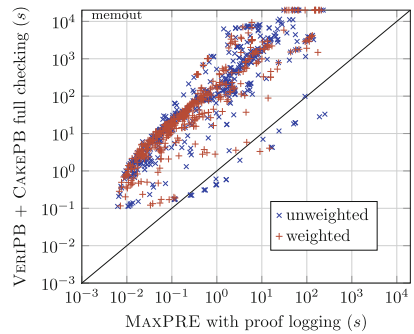


Fig. 7. MAXPRE vs. combined proof checking running time.

To answer the first question, in Fig. 6 we compare MAXPRE with and without proof logging. In total, 1081 instances were successfully preprocessed by MAXPRE without proof logging. With proof logging enabled, 8 fewer instances were preprocessed due to either time- or memory-outs. For the successfully preprocessed instances, the geometric mean of the proof logging overhead is 46% of the

running time, and 95% of the instances were preprocessed with proof logging in at most twice the time required without proof logging.

Our comparison between proof generation and proof checking is based on the 1073 instances for which preprocessing with proof logging was successful. Out of these, 1021 instances were successfully checked and elaborated by VERIPB. For 991 instances the verdicts were confirmed by the formally verified proof checker CAKEPBWCNF, with the remaining instances being time- or memory-outs. This shows the practical viability of our approach, as the vast majority of preprocessing proofs were checked within the resource limits.

A scatter plot comparing the running time of MAXPRE with proof logging enabled against the combined checking process is shown in Fig. 7. For the combined checking time, we only consider the instances that have been successfully checked by CAKEPBWCNF. In the geometric mean, the time for the combined verified checking pipeline of VERIPB elaboration followed by CAKEPBWCNF checking is $113\times$ the preprocessing time of MAXPRE. A general reason for this overhead is that the preprocessor has more MaxSAT application-specific context than the pseudo-Boolean checker, so the preprocessor can log proof steps without performing the actual reasoning while the checker must ensure that those steps are sound in an application-agnostic way. An example for this is reification: as the preprocessor knows its reification variables are fresh, it can easily emit redundancy steps that witness on those variables; but the checker has to verify freshness against its own database. Similar behaviour has been observed in other applications of pseudo-Boolean proof logging [27, 37].

To analyse further the causes of proof checking overhead, we also compared VERIPB to CAKEPBWCNF. The checking of the elaborated kernel proof with CAKEPBWCNF is $6.7\times$ faster than checking and elaborating the augmented proof with VERIPB. This suggests that the bottleneck for proof checking is VERIPB; VERIPB *without* elaboration is about $5.3\times$ slower than CAKEPBWCNF. As elaboration is a necessary step before running CAKEPBWCNF, improving the performance of VERIPB would benefit the performance of the pipeline as a whole. One specific feature that seems desirable would be to augment RUP rule applications with LRAT-style hints [16], so that VERIPB would not need to perform unit propagation to elaborate RUP steps to cutting planes derivations. Though these types of engineering challenges are important to address, they are beyond the scope of the current paper and we have to leave them as future work.

6 Conclusion

In this work, we show how to use pseudo-Boolean proof logging to certify correctness of the MaxSAT preprocessing phase, extending previous work for the main solving phase in unweighted model-improving solvers [72] and general core-guided solvers [4]. As a further strengthening of previous work, we present a fully formally verified toolchain which provides end-to-end verification of correctness.

In contrast to SAT solving, there is a rich variety of techniques in maximum satisfiability solving, and it still remains to design pseudo-Boolean proof logging

methods for general, weighted, model-improving MaxSAT solvers [21, 47, 62] and *implicit hitting set (IHS)* MaxSAT solvers [18, 19] with *abstract cores* [3]. Nevertheless, our work adds further weight to the conclusion that pseudo-Boolean reasoning seems like a very promising foundation for MaxSAT proof logging. We are optimistic that this work is another step on the path towards general adoption of proof logging in the context of SAT-based optimization.

Acknowledgments. This work has been financially supported by the University of Helsinki Doctoral Programme in Computer Science DoCS, the Research Council of Finland under grants 342145 and 346056, the Swedish Research Council grants 2016-00782 and 2021-05165, the Independent Research Fund Denmark grant 9040-00389B, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and by A*STAR, Singapore. Part of this work was carried out while some of the authors participated in the extended reunion of the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley. We also acknowledge useful discussions at the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*. The computational experiments were enabled by resources provided by LUNARC at Lund University.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C., Schweitzer, P.: An introduction to certifying algorithms. *IT - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* **53**(6), 287–293 (2011)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Milano, M. (eds.) *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*. LNCS, vol. 7514, pp. 86–101. Springer, Cham (2012). https://doi.org/10.1007/978-3-642-33558-7_9
3. Berg, J., Bacchus, F., Poole, A.: Abstract cores in implicit hitting set MaxSat solving. In: Pulina, L., Seidl, M. (eds.) *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*. LNCS, vol. 12178, pp. 277–294. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_20
4. Berg, J., Bogaerts, B., Nordström, J., Oertel, A., Vandesande, D.: Certified core-guided MaxSAT solving. In: Pientka, B., Tinelli, C. (eds.) *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*. LNCS, vol. 14132, pp. 1–22. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38499-8_1
5. Berg, J., Järvisalo, M.: Unifying reasoning and core-guided search for maximum satisfiability. In: Calimeri, F., Leone, N., Manna, M. (eds.) *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*. LNCS, vol. 11468, pp. 287–303. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19570-0_19
6. Berg, J., Saikko, P., Järvisalo, M.: Subsumed label elimination for maximum satisfiability. In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*. FAIA, vol. 285, pp. 630–638. IOS Press (2016)
7. Biere, A.: Tracecheck (2006). <http://fmv.jku.at/tracecheck/>

8. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, 2nd edn., vol. 336. IOS Press, February 2021
9. Bogaerts, B., Gocht, S., McCreesh, C., Nordström, J.: Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.* **77**, 1539–1589 (2023). Preliminary version in AAAI 2022
10. Bogaerts, B., McCreesh, C., Myreen, M.O., Nordström, J., Oertel, A., Tan, Y.K.: Documentation of VeriPB and CakePB for the SAT competition 2023, March 2023. Available at <https://satcompetition.github.io/2023/checkers.html>
11. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. *Artif. Intell.* **171**(8–9), 606–618 (2007)
12. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. *IEEE Trans. Syst. Man Cybern. Part B (Cybern.)* **34**(1), 52–59 (2004)
13. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010). LNCS, vol. 6175, pp. 44–57. Springer, Cham (2010). https://doi.org/10.1007/978-3-642-14186-7_6
14. Buss, S.R., Nordström, J.: Proof complexity and SAT solving. In: Biere et al. [8], Chap. 7, pp. 233–350
15. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. *Discret. Appl. Math.* **18**(1), 25–38 (1987)
16. Cruz-Filipe, L., Heule, M.J.H., Hunt Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (eds.) Proceedings of the 26th International Conference on Automated Deduction (CADE-26). LNCS, vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
17. Cruz-Filipe, L., Marques-Silva, J.P., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). LNCS, vol. 10205, pp. 118–135. Springer, Cham (2017). https://doi.org/10.1007/978-3-662-54577-5_7
18. Davies, J., Bacchus, F.: Solving MAXSAT by solving a sequence of simpler SAT instances. In: Lee, J. (eds.) Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011). LNCS, vol. 6876, pp. 225–239. Springer, Cham (2011). https://doi.org/10.1007/978-3-642-23786-7_19
19. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 166–181. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_13
20. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005). LNCS, vol. 3569, pp. 61–75. Springer, Cham (2005). https://doi.org/10.1007/11499107_5
21. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. *J. Satisfiability Boolean Model. Comput.* **2**(1–4), 1–26 (2006)
22. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using pseudo-Boolean reasoning. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), pp. 1486–1494, February 2020

23. Filmus, Y., Mahajan, M., Sood, G., Vinyals, M.: MaxSAT resolution and subcube sums. In: Pulina, L., Seidl, M. (eds.) Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020). LNCS, vol. 12178, pp. 295–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_21
24. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. Ph.D. thesis, University of Pennsylvania (1995)
25. Gimpel, J.F.: A reduction technique for prime implicant tables. In: Proceedings of the 5th Annual Symposium on Switching Circuit Theory and Logical Design, (SWCT 1964), pp. 183–191. IEEE Computer Society (1964)
26. Gocht, S.: Certifying correctness for combinatorial algorithms by using pseudo-Boolean reasoning, Ph.D. thesis, Lund University, June 2022. <https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu>
27. Gocht, S., Martins, R., Nordström, J., Oertel, A.: Certified CNF translations for pseudo-Boolean solving. In: Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 16:1–16:25, August 2022
28. Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., Trimble, J.: Certifying solvers for clique and maximum common (connected) subgraph problems. In: Simonis, H. (eds.) Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP 2020). LNCS, vol. 12333, pp. 338–357. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_20
29. Gocht, S., McCreesh, C., Myreen, M.O., Nordström, J., Oertel, A., Tan, Y.K.: End-to-end verification for subgraph solving. In: Proceedings of the 368h AAAI Conference on Artificial Intelligence (AAAI 2024), pp. 8038–8047, February 2024
30. Gocht, S., McCreesh, C., Nordström, J.: Subgraph isomorphism meets cutting planes: solving with certified solutions. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020), pp. 1134–1140, July 2020
31. Gocht, S., McCreesh, C., Nordström, J.: An auditable constraint programming solver. In: Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, pp. 25:1–25:18, August 2022
32. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021), pp. 3768–3777, February 2021
33. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2003), pp. 886–891, March 2003
34. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Yang, H. (eds.) Proceedings of the 26th European Symposium on Programming (ESOP 2017). LNCS, vol. 10201, pp. 584–610. Springer, Cham (2017). https://doi.org/10.1007/978-3-662-54434-1_22
35. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2013), pp. 181–188, October 2013
36. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Bonacina, M.P. (eds.) Proceedings of the 24th International Conference on Automated Deduction (CADE-24). LNCS, vol. 7898, pp. 345–359. Springer, Cham (2013). https://doi.org/10.1007/978-3-642-38574-2_24

37. Hoen, A., Oertel, A., Gleixner, A., Nordström, J.: Certifying MIP-based presolve reductions for 0–1 integer linear programs. In: Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2024), May 2024, to appear
38. Ignatiev, A., Morgado, A., Marques-Silva, J.: RC2: an efficient MaxSAT solver. *J. Satisfiability Boolean Model. Comput.* **11**(1), 53–64 (2019)
39. Ihalainen, H., Berg, J., Järvisalo, M.: Clause redundancy and preprocessing in maximum satisfiability. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR 2022). LNCS, vol. 13385, pp. 75–94. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_6
40. Ihalainen, H., et al.: Certified MaxSAT preprocessing (2024). <https://arxiv.org/abs/2404.17316>. Full-length version
41. Ihalainen, H., et al.: Experimental Repository for “Certified MaxSAT Preprocessing”, February 2024. <https://doi.org/10.5281/zenodo.10630852>
42. Jabs, C., Berg, J., Ihalainen, H., Järvisalo, M.: Preprocessing in SAT-based multi-objective combinatorial optimization. In: Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 280, pp. 18:1–18:20 (2023)
43. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010). LNCS, vol. 6015, pp. 129–144. Springer, Cham (2010). https://doi.org/10.1007/978-3-642-12002-2_10
44. Korhonen, T., Berg, J., Saikko, P., Järvisalo, M.: MaxPre: an extended MaxSAT preprocessor. In: Gaspers, S., Walsh, T. (eds.) Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT 2017). LNCS, vol. 10491, pp. 449–456. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_28
45. Larrosa, J., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A framework for certified Boolean branch-and-bound optimization. *J. Autom. Reason.* **46**(1), 81–102 (2011)
46. Le Berre, D.: Exploiting the real power of unit propagation lookahead. *Electron. Notes Discrete Math.* **9**, 59–80 (2001)
47. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. *J. Satisfiability Boolean Model. Comput.* **7**, 59–64 (2010)
48. Li, C.M.: Integrating equivalency reasoning into Davis-Putnam procedure. In: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, pp. 291–296. AAAI Press/The MIT Press (2000)
49. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of Boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) 8th International Haifa Verification Conference (HVC 2012), Revised Selected Papers. LNCS, vol. 7857, pp. 102–117. Springer, Cham (2013). https://doi.org/10.1007/978-3-642-39611-3_14
50. MaxPre 2: MaxSAT preprocessor. <https://bitbucket.org/coreo-group/maxpre2>
51. MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. <https://maxsat-evaluations.github.io/>
52. MaxSAT evaluation 2023, July 2023. <https://maxsat-evaluations.github.io/2023>
53. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. *Comput. Sci. Rev.* **5**(2), 119–161 (2011)

54. McIlree, M., McCreesh, C.: Proof logging for smart extensional constraints. In: Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 280, pp. 26:1–26:17, August 2023
55. McIlree, M., McCreesh, C., Nordström, J.: Proof logging for the circuit constraint. In: Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2024), May 2024, to appear
56. Morgado, A., Heras, F., Marques-Silva, J.: Improvements to core-guided binary search for MaxSAT. In: Cimatti, A., Sebastiani, R. (eds.) Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012). LNCS, vol. 7317, pp. 284–297. Springer, Cham (2012). https://doi.org/10.1007/978-3-642-31612-8_22
57. Morgado, A., Ignatiev, A., Bonet, M.L., Marques-Silva, J.P., Buss, S.R.: DRMaxSAT with MaxHS: first contact. In: Janota, M., Lynce, I. (eds.) Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT 2019). LNCS, vol. 11628, pp. 239–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_17
58. Morgado, A., Marques-Silva, J.: On validating Boolean optimizers. In: Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2011), pp. 924–926 (2011)
59. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.* **24**(2–3), 284–315 (2014)
60. Ostrowski, R., Grégoire, É., Mazure, B., Saïs, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 185–199. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_13
61. Paxian, T., Raiola, P., Becker, B.: On preprocessing for weighted MaxSAT. In: Henglein, F., Shoham, S., Vizek, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 556–577. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_25
62. Paxian, T., Reimer, S., Becker, B.: Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In: Beyersdorff, O., Wintersteiger, C. (eds.) Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018). LNCS, vol. 10929, pp. 37–53. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_3
63. Py, M., Cherif, M.S., Habet, D.: Towards bridging the gap between SAT and MaxSAT refutations. In: Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2020), pp. 137–144, November 2020
64. Py, M., Cherif, M.S., Habet, D.: A proof builder for Max-SAT. In: Li, C.M., Manyá, F. (eds.) Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT 2021). LNCS, vol. 12831, pp. 488–498. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_33
65. Py, M., Cherif, M.S., Habet, D.: Proofs and certificates for Max-SAT. *J. Artif. Intell. Res.* **75**, 1373–1400 (2022)
66. The International SAT Competitions web page. <http://www.satcompetition.org>
67. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_6
68. Subbarayan, S., Pradhan, D.K.: NiVER: non-increasing variable elimination resolution for preprocessing SAT instances. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT

2004. LNCS, vol. 3542, pp. 276–291. Springer, Heidelberg (2005). https://doi.org/10.1007/11527695_22
69. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in CakeML. *Int. J. Softw. Tools Technol. Transf.* **25**, 167–184 (2023). Preliminary version in TACAS 2021
 70. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: The verified CakeML compiler backend. *J. Funct. Program.* **29**, e2:1–e2:57 (2019)
 71. Van Gelder, A.: Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.* **43**(1), 239–253 (2005)
 72. Vandesande, D., De Wulf, W., Bogaerts, B.: QMaxSATpb: a certified MaxSAT solver. In: Gottlob, G., Inclezan, D., Maratea, M. (eds.) *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR 2022)*. LNCS, vol. 13416, pp. 429–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15707-3_33
 73. VeriPB: Verifier for pseudo-Boolean proofs. <https://gitlab.com/MIAOresearch/software/VeriPB>
 74. Wetzler, N., Heule, M.J.H., Hunt Jr., W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
 75. Zabih, R., McAllester, D.A.: A rearrangement search strategy for determining propositional satisfiability. In: *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)*, pp. 155–160. AAAI Press/The MIT Press (1988)

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

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





A Formal Model to Prove Instantiation Termination for E-matching-Based Axiomatisations

Rui Ge^(✉), Ronald Garcia, and Alexander J. Summers

Department of Computer Science, University of British Columbia,
Vancouver, BC, Canada
{rge,rxg}@cs.ubc.ca, alex.summers@ubc.ca

Abstract. SMT-based program analysis and verification often involve reasoning about program features that have been specified using quantifiers; incorporating quantifiers into SMT-based reasoning is, however, known to be challenging. If quantifier instantiation is not carefully controlled, then runtime and outcomes can be brittle and hard to predict. In particular, uncontrolled quantifier instantiation can lead to unexpected incompleteness and even non-termination. E-matching is the most widely-used approach for controlling quantifier instantiation, but when axiomatisations are complex, even experts cannot tell whether or not their use of E-matching guarantees completeness or termination.

This paper presents a new formal model that facilitates the proof, once and for all, that giving a complex E-matching-based axiomatisation to an SMT solver such as Z3 or cvc5, cannot cause non-termination. Key to our technique is an operational semantics for solver behaviour that models how the E-matching rules common to most solvers are used to determine when quantifier instantiations are enabled, but abstracts over irrelevant details of individual solvers. We demonstrate the effectiveness of our technique by presenting a termination proof for a set theory axiomatisation adapted from those used in the Dafny and Viper verifiers.

Keywords: SMT solving · Quantifiers · Termination proofs · E-matching

1 Introduction

SMT-based program analysis and verification have advanced dramatically in the past two decades. These advances have been partly fuelled by major improvements in SAT and SMT solving techniques, as well as their implementations in state-of-the-art solvers such as Z3 [22] and cvc5 [2]. Leveraging these advances in SMT, a huge number of program analysis and verification tools have been based on SMT, including for example Dafny [17], Why3 [12] and Viper [24].

Such tools must translate a wide range of problem features into SMT queries that model these domain-specific concerns. While some theories relevant to problem features (e.g. linear arithmetic [22]) are natively supported by SMT solvers, most problem features must be modelled by *axiomatisation*.

Axiomatising problem features involves introducing uninterpreted sorts, uninterpreted functions on these sorts, and (crucially) *quantifiers*¹ that define the intended meaning of these features. For instance, one can model sets of integers by introducing a sort *Set* for sets, uninterpreted functions *member* and *diff* to represent set membership and set difference respectively, and quantifiers such as $\forall s_1, s_2 : \text{Set}, x : \text{Int}. \text{member}(x, s_2) \rightarrow \neg \text{member}(x, \text{diff}(s_1, s_2))$.

Such modelling to SMT is expressive, but makes heavy use of quantifiers that must be instantiated during SMT solving. But quantifier instantiation in SMT notoriously presents notable challenges, potentially causing slow performance and even non-termination, as well as unexpectedly-failing proofs [4, 19]. Worse still, latent quantifier instantiation issues may not surface on all runs, but cause a “butterfly effect” [16], meaning that unrelated changes to an input problem may lead to substantial changes in solver behaviour along these lines.

To manage these issues, solvers allow quantifiers to be annotated with instantiation *triggers* (a.k.a. instantiation *patterns*). Triggers specify (possibly multiple) shapes of ground terms that must be *known* (occur in the current proof context, modulo known equalities) to enable a quantifier instantiation. This method of guiding quantifier instantiation is referred to as *E-matching* [8, 25] and is supported by virtually all modern SMT solvers.

However, selecting appropriate triggers is an art. The choice requires expertise in managing a fine balance: not too restrictive, to avoid insufficient quantifier instantiations for proofs, and not too permissive, to prevent excessive instantiations. Subtle issues can easily lead to the same hard-to-debug problems even for the most talented of SMT artists [16, 19], and even when successful it is unclear how one can *know* that the chosen triggers are guaranteed to work in the future.

The ideal aim is to achieve both instantiation completeness and instantiation termination. *Instantiation completeness* means that all necessary quantifier instantiations for a proof can be made by the solver. *Instantiation termination* means that the solver will never endlessly explore infinitely many quantifier instantiations. In this paper, we focus on instantiation termination.²

Failures of instantiation termination stem from *matching loops*: the problematic scenario of a quantifier instantiation (possibly indirectly) leading to learning new terms that cause further instantiations of the same quantifier, potentially creating an endless loop. Matching loops *can* cause non-termination, but (problematically, for debugging) may only do so on some runs (in case heuristics in the solver arrive at the facts necessary to complete a proof “in time”).

¹ We use the term *quantifier* (also) as a synonym for quantified formula.

² Instantiation termination can be trivially achieved by pathological trigger choices that prevent all instantiations (similar to proving a function terminating under a false precondition). However, such axiomatisations are not useful (or used) in practice.

Our paper enables proving that matching loops have been avoided altogether. We present a high-level formal model of E-matching-based quantifier instantiation that suffices to prove *once and for all* that a given set of trigger-annotated quantifiers, when combined with *any possible* ground facts, guarantees instantiation termination, thereby ensuring the absence of matching loops. Our model is designed to be broadly applicable because it models the core E-matching rules common to most solvers, but abstracts over implementation details where individual solvers make different choices. Our model enables formal termination proofs based on familiar concepts from program reasoning, with manageable complexity, allowing axiomatisation practitioners to independently construct these proofs and confidently seek terminating responses to ground theory queries.

Our main technical contributions are as follows:

1. We develop a formal model for reasoning about instantiation termination in E-matching-based axiomatisations. The model abstracts from solver implementation details but accounts for the essential features necessary for rigorous instantiation termination proofs.
2. We validate the practical utility of our formal model by using it to prove instantiation termination of a challenging set theory axiomatisation adapted from the cores of those used in the Dafny and Viper verifiers.
3. We outline a methodology for constructing instantiation termination proofs using our model. Our methodology involves classifying quantifiers according to certain characteristics, using these to incrementally define and refine a progress measure that eventually supports the whole axiomatisation.

Our research draws inspiration from Dross et al.'s [11] prior formalism for quantifier instantiation via E-matching. To the best of our knowledge, their work represents the sole formal attempt in this space before ours. However, we find their formalism incompatible with our goals: we elaborate on this point in Sect. 5.

Full details and supporting proofs are available in our technical report (TR hereafter) [13].

2 Problem Statement

We begin with a basic grounding in E-matching, and use this to lay out the most important challenges a formal model needs to address to be useful in practice.

2.1 Quantifier Instantiation via E-matching

Quantifiers are crucial for effectively modelling external problem features as an SMT problem. However, when determining whether such a first-order problem is satisfiable, an SMT solver must contend with quantifiers ranging over infinite sorts. A successful proof will (and need) only involve finitely many instantiations of the quantifiers, but selecting these is in general undecidable. Most solvers provide *E-matching* as the main means of guiding instantiation.

E-matching requires each quantifier to be associated with instantiation *triggers* (a.k.a. instantiation *patterns*). Triggers consist of terms containing the quantified variables, and prescribe that instantiations should only be made when ground terms of matching shape(s) arise in the current proof search.

During a proof search, SMT solvers maintain and update the currently-known ground terms and (dis)equalities on them in an efficient congruence-closure data structure called an *E-graph*. This information enables *E-matching* [21, 25]—matching modulo currently-known equalities—of known terms against quantifier triggers, which enables new instantiations, and of potential instantiations against previous ones, which prevents redundant instantiations.

Example 1. Consider the set theory axiom presented early in Sect. 1, now annotated with triggers (written comma-separated inside square brackets)³:

$$\forall s_1, s_2, x. [diff(s_1, s_2), member(x, s_2)] member(x, s_2) \rightarrow \neg member(x, diff(s_1, s_2))$$

The trigger consists of two terms, $diff(s_1, s_2)$ and $member(x, s_2)$; a multi-term trigger prescribes that terms matching *all* (here, both) patterns must be known for some instantiation of the quantified variables. If so, the corresponding instantiation of the quantifier *itself* will be made: the instantiated quantifier body⁴ will be treated as a newly-derived fact (typically, a *clause*), and the solver will also record that this instantiation has been made (to avoid doing so again).

Suppose that an E-graph represents the congruence closure of the facts: $member(t, a) = \top$, $diff(b, c) \neq b$ and $a = c$. E-matching will find a successful match against the trigger above; although it might seem that there is no consistent pair of terms here, the equality $a = c$ means that (modulo equalities) we can consider the terms $member(t, a)$ and $diff(b, a)$ as known in the E-graph, which match the triggers under the instantiation $s_1 \mapsto b$, $s_2 \mapsto a$ and $x \mapsto t$. The corresponding instantiation of the quantifier body yields $\neg member(t, a) \vee \neg member(t, diff(b, a))$. Subsequently, the same quantifier cannot be instantiated with e.g. $s_1 \mapsto b$, $s_2 \mapsto c$ and $x \mapsto t$ since, again modulo equalities, this is an equivalent instantiation.

Example 2. Consider a variant of the previous quantifier, modified with a different trigger, and in the context of a different E-graph that represents instead the congruence closure of the facts: $member(t, a) = \top$ and $member(t, b) = \top$.

$$\forall s_1, s_2, x. [member(x, s_1), member(x, s_2)] member(x, s_2) \rightarrow \neg member(x, diff(s_1, s_2))$$

Now four instantiations are enabled: one for each pair of *member* applications in our current model (and E-graph): e.g. instantiating $s_1 \mapsto a$, $s_2 \mapsto b$ and $x \mapsto t$ or $s_1 \mapsto b$, $s_2 \mapsto a$ and $x \mapsto t$. All four will be made: they are different choices since we don't know that $a = b$. The second, for example, causes the new clause (rewritten as a disjunction) $\neg member(t, a) \vee \neg member(t, diff(b, a))$ to be assumed. This doesn't change the E-graph (which is populated only by assumed *literals*);

³ For brevity, sorts on quantified variables are omitted in this example and hereafter.

⁴ *Quantifier body* refers to the subformula that falls within the scope of a quantifier.

clauses are kept separately in the prover state. However, case-splitting on this clause may lead to the literal $\neg member(t, diff(b, a))$ being added. At this point, five *new* quantifier instantiations will be enabled; the number of pairs of *member* applications has increased. In fact, by alternately instantiating this quantifier and case-splitting on newly-learned clauses, we can uncover new instantiations indefinitely, in a so-called *matching loop*.

These first examples show that the choice of triggers affects instantiation behaviour, and that modelling instantiations requires considering not only initial terms, but also facts learned during proof search and case-splitting choices.

Example 3. Consider the following “subset elimination” axiom (from the set theory axiomatisation we tackle later) with nested quantifiers:

$$\forall s_1, s_2. [subset(s_1, s_2)] \quad subset(s_1, s_2) \rightarrow \\ (\forall x. [member(x, s_1)][member(x, s_2)] \quad member(x, s_1) \rightarrow member(x, s_2))$$

The inner quantifier has *two* triggers, defining *alternative* conditions for instantiation (a term of either shape is sufficient). Note that these triggers depend on the outer-quantified variables s_1 and s_2 , and thus their instantiations.

Instantiating an outer quantifier expands the current quantifiers for instantiations. In this example, instantiating the outer quantifier ($\forall s_1, s_2. \dots$) results in a clause that includes a copy of the inner quantifier ($\forall x. \dots$); case-splitting on this clause can cause the copy to be assumed, effectively adding one more quantifier for future potential instantiations. As such, the instantiation of outer quantifiers *dynamically* introduces new quantifiers, adding complexity to establishing termination arguments—one must be able to identify and predict the quantifiers (and their instantiations) that will be dynamically introduced.

2.2 Objectives for a Formal Model of E-matching

Given the difficulty of choosing quantifier triggers and *knowing* that their instantiations can *never* continue forever, our objective is to provide formal and usable means of proving such E-matching *termination proofs* once-and-for-all. Rather than attempt to capture the precise behaviour of a specific solver and its configuration, we want a model that abstracts over the behaviours of *any* reasonable implementation of E-matching, while still being sufficiently precise for the proofs to work and be reasonable to construct in practice.

The design of a model for E-matching must address multiple challenges:

1. How should (intermediate) solver states and the transitions between them be modelled, avoiding over-fitting to specific solver choices while retaining clear and pertinent information suitable for understandable proofs?
2. How should equality-related information and reasoning be captured, given their central nature (for defining enabled E-matches) but the complexities of the data structures employed in real implementations?
3. How can nested quantifiers (cf. Example 3), when instantiations can introduce new quantifiers on the fly, be supported?

4. How can we make the model extensible to more-complex future applications (e.g. axiomatisations whose termination depends on theory reasoning)?
5. How can a formal model enable formal proofs with manageable complexity?

We present our model, designed to address these challenges in the next section; we demonstrate its applicability for termination proofs in Sect. 4.

3 An Operational Semantics for E-matching

We develop our formal model in the style of a *small-step operational semantics*, a popular choice for programming languages. In this operational style, states represent intermediate points of a proof search, while transitions represent solver steps; non-determinism abstracts over choices specific solvers make. With this design, our desired notion of instantiation termination can be recast as a familiar style of termination proof, albeit against a semantics with novel core details.

3.1 Preliminaries

Our syntax for formulas is based around a generalisation of conjunctive normal form, used internally in SMT algorithms; we assume all formulas are pre-converted to this form (existential quantifiers are eliminated by Skolemisation).

Definition 1 (Formula Syntax). *We assume a pre-defined set of atoms⁵, including equalities on terms $t_1 = t_2$. A (simple) literal l is either an atom or its negation. The grammars of extended literals ϕ , extended clauses C and extended conjunctive normal form (ECNF) formulas A are as follows:*

$$\phi ::= l \mid (\forall \vec{x}. [\vec{T}]A)^{\sharp\alpha} \quad C ::= \phi \mid C \vee C \quad A ::= C \mid A \wedge A$$

Here, $(\forall \vec{x}. [\vec{T}]A)^{\sharp\alpha}$ denotes a tagged quantifier: the (possibly-multiple) variables \vec{x} are bound, the (possibly-multiple) trigger sets \vec{T} are each marked with square brackets and positioned before the quantifier body A , and $\sharp\alpha$ is a tag used to uniquely identify this particular quantifier (see also Sect. 3.6).

As presented in Example 1, a trigger set T is a (non-empty) set of terms, written comma-separated. There are additional requirements: each trigger set must contain each quantified variable at least once, and each term must contain at least one quantified variable. Furthermore, each term must contain at least one uninterpreted function application and no interpreted function symbols such as equalities. These restrictions are common for SMT solvers.

When quantifier tags are not relevant, we omit them for brevity.

⁵ The pre-defined atoms come from the first-order signature of the problem in question.

3.2 States

As illustrated in Examples 1 and 2, both case-splitting and quantifier instantiation steps are crucial to our problem; we define our semantics around these two kinds of transitions. Furthermore, we must abstractly capture information relevant for deciding E-matching questions, tracking in particular which terms and equalities are known (modulo currently known equalities), and which quantifier instantiations have already been made.

Definition 2 (States). *States $s \in STATE$ are defined as follows:*

$$s ::= \langle W, A, E \rangle \mid \diamond \mid \perp$$

where \diamond and \perp are distinguished symbols for saturated and inconsistent states, W (the current quantifiers) is a set of tagged quantifiers, A (the current clauses) is a set of extended clauses, and E (the current E-state) is explained below.

For simple applications of our semantics, the set of current quantifiers remains fixed, but for problems with nested quantifiers (e.g. Example 3), it may grow as a solver runs. As we show, which instantiations are immediately enabled is definable in terms of both the current quantifiers and the current E-state. The current clauses, on the other hand, through case-splitting, introduce new quantifiers to the current quantifiers and generate new literals for the E-state; new extended clauses may be added as a consequence of quantifier instantiations.

The inconsistent and saturated states represent two different termination conditions for traces in our semantics: the former due to logical inconsistency, and the latter due to all quantifier instantiations having been exhausted.

3.3 E-interfaces

Each solver maintains its own implementation of E-graphs to efficiently represent and query the currently-known ground terms modulo congruences and known equalities. Rather than formalising such an implementation, we devise an abstraction called an *E-interface*, capturing the operations and expected mathematical properties of E-graph implementations.

Definition 3 (E-interface Judgements). *An E-interface E^I is a set of equalities and disequalities on terms.⁶ We write $E^I \Vdash_{\text{kn}} t$ to express that the ground term t is known in the E-interface E^I ; we write $E^I \Vdash t_1 \sim t_2$ to express that the ground terms t_1 and t_2 are known equal in E^I . These two judgements are (mutually recursively) defined by (the least fixed-point of) the derivation rules:*

$$\frac{t_1 \sim t_2 \in E^I}{E^I \Vdash t_1 \sim t_2} \text{(EQ-IN)} \qquad \frac{E^I \Vdash t_2 \sim t_1}{E^I \Vdash t_1 \sim t_2} \text{(EQ-SYM)}$$

$$\frac{E^I \Vdash t_1 \sim t_2 \quad E^I \Vdash t_2 \sim t_3}{E^I \Vdash t_1 \sim t_3} \text{(EQ-TRAN)} \qquad \frac{E^I \Vdash_{\text{kn}} t}{E^I \Vdash t \sim t} \text{(EQ-KN-REFL)}$$

⁶ A positive or negative non-equational literal, P , is added to the E-interface via $P = \top$ or $P = \perp$, respectively; $\top \neq \perp$ is preloaded into all E-interfaces.

$$\frac{\frac{E^I \Vdash t_i \sim t'_i \quad E^I \Vdash_{\text{kn}} g(t_1, \dots, t_i, \dots, t_n)}{E^I \Vdash g(t_1, \dots, t_i, \dots, t_n) \sim g(t_1, \dots, t'_i, \dots, t_n)} \text{(EQ-KN-SUB)}}{\frac{E^I \Vdash t_1 \sim t_2 \quad E^I \Vdash_{\text{kn}} g(\dots, t_i, \dots)}{E^I \Vdash_{\text{kn}} t_1} \text{(KN-EQ)} \quad \frac{E^I \Vdash_{\text{kn}} g(\dots, t_i, \dots)}{E^I \Vdash_{\text{kn}} t_i} \text{(KN-SUB)}}$$

The judgement $E^I \Vdash t_1 \not\sim t_2$ represents t_1 and t_2 being known disequal in E^I ; the judgement $E^I \Vdash \perp$ represents that E^I is inconsistent (in the logical sense); cf. App. A of the TR.

E-interfaces are equivalent if they agree on these judgements in all cases. When a proof step adds new literals, we must be able to extend our E-interfaces.

Definition 4 (E-interface Extension). For a set of equality and disequality literals L , the update of an E-interface E^I with L , denoted $E^I \triangleleft L$, is a minimal E-interface which satisfies all E-interface judgements that E^I does, while also satisfying $E^I \Vdash l$ for all $l \in L$.

We call a set of terms a *basis* of E^I if each element is a representative of a different equivalence class⁷ induced by the $E^I \Vdash t_1 \sim t_2$ relation on the terms known in E^I . As we shall see in the next subsection, equivalence classes are relevant for defining which quantifier instantiations can be made after which.

3.4 E-histories, E-states, E-matching

As illustrated in Example 1, E-matching against triggers does not suffice to determine whether a quantifier instantiation should be considered *enabled*; we must also determine whether the instantiation is considered redundant given *previous* ones. We record previous instantiations using our next formal ingredient:

Definition 5 (E-histories and E-states). An E-history E^H is a set of pairs (each denoted $(\sharp\alpha : \vec{r})$) in our formalism: the first element is a tag (identifying a quantifier), and the second is a vector of ground terms (representing an instantiation of the corresponding quantifier).

An E-state (cf. Definition 2) E is a pair (E^I, E^H) of E-interface and E-history.

Recall that E-states are a part of the states in our formalism. E-states consist of an E-interface component, which captures the current known terms and equality information, and an E-history component, which records the history of instantiations, in particular representing sufficient information to reject redundant instantiations.

Definition 6 (History-Enabled E-matches). Given a candidate match pair $(\sharp\alpha : \vec{r})$ (of tag $\sharp\alpha$ and vector of terms \vec{r}), the E-state E enables $(\sharp\alpha : \vec{r})$, written $E \Vdash_{\text{hist}} (\sharp\alpha : \vec{r})$, if: for every instantiation pair $(\sharp\alpha : \vec{r}') \in E^H$, at least one of the pointwise equalities $r_i \sim r'_i$ is not known in E^I .

⁷ What we refer to as an *equivalence class* in this paper is also known as a *congruence class* in the literature: an equivalence class modulo known equalities.

Example 4. Revisiting Example 1, suppose the tag of the quantifier is $\sharp\tau$, E is the E-state whose E-interface component contains the example literals. The first instantiation $s_1 \mapsto b$, $s_2 \mapsto a$ and $x \mapsto t$ is represented in our formal model by adding $(\sharp\tau : (b, a, t))$ to the E-history, resulting in a new E-state, say E' . The second candidate match $s_1 \mapsto b$, $s_2 \mapsto c$ and $x \mapsto t$ is not enabled in E' since the three pointwise equalities between instantiated terms are all known in E' .

With the help of the above ingredients, we formally characterise E-matching:

Definition 7 (E-matching). *For a given state $\langle W, A, E \rangle$, the judgement $\langle W, A, E \rangle \vdash_{\text{match}} (\forall \vec{x}. \overrightarrow{[T]} A')^{\sharp\alpha} \triangleleft \vec{r}$ defines which instantiations (using terms \vec{r}) of which quantifiers $(\forall \vec{x}. \overrightarrow{[T]} A')^{\sharp\alpha}$ are enabled by E-matching rules, as follows:*

$$\frac{(\forall \vec{x}. \overrightarrow{[T]} A')^{\sharp\alpha} \in W \quad \vec{t} \text{ is one trigger set of } \overrightarrow{[T]} \quad E^I \Vdash_{\text{kn}} \vec{t} [\vec{r}/\vec{x}] \quad E \Vdash_{\text{hist}} (\sharp\alpha : \vec{r})}{\langle W, A, E \rangle \vdash_{\text{match}} (\forall \vec{x}. \overrightarrow{[T]} A')^{\sharp\alpha} \triangleleft \vec{r}}$$

We write $\langle W, A, E \rangle \not\vdash_{\text{match}}$ to mean no instantiations are enabled in this state.

E-matching \vdash_{match} requires (1) a quantifier in the current state, (2) a trigger set \vec{t} with replacement terms \vec{r} for quantified variables \vec{x} to be known in E^I , and (3) that this potential match is enabled by the E-state E . Note that (2) implies the terms \vec{r} to match against the quantified variables of one trigger set \vec{t} to be known in the current E-interface E^I .

3.5 State Transitions

The last main ingredient of our formal model is the definition of state transitions.

Definition 8 (State Transitions). *The (single step) state transition relation $\longrightarrow \subseteq \text{STATE} \times \text{STATE}$ is defined by the union of the following cases:*

$$\frac{\emptyset \subset \Phi \subseteq \{\phi_i \mid C \in A; W_1, E_1^I \not\vdash_{\text{sat}} C; C \text{ is } \dots \vee \phi_i \vee \dots\} \quad W_2 = W_1 \cup \text{filter}_{\forall}(\Phi) \quad E_2^I = E_1^I \triangleleft \text{filter}_{\text{lit}}(\Phi) \quad E_2^H = E_1^H}{\langle W_1, A, E_1 \rangle \longrightarrow \langle W_2, A, E_2 \rangle} \text{(SPLIT)}$$

$$\frac{E^I \Vdash \perp}{\langle W, A, E \rangle \longrightarrow \perp} \text{(BOT)}$$

$$\frac{E^I \not\vdash \perp \quad W, E^I \Vdash_{\text{sat}} C \text{ for every } C \in A \quad \langle W, A, E \rangle \not\vdash_{\text{match}}}{\langle W, A, E \rangle \longrightarrow \diamond} \text{(SAT)}$$

$$\frac{\langle W_1, A_1, E_1 \rangle \vdash_{\text{match}} (\forall \vec{x}. \overrightarrow{[T]} A_{11})^{\sharp\alpha} \triangleleft \vec{r} \quad A_{12} = A_{11} [\vec{r}/\vec{x}] \quad A'_{12} = \text{filter}_{\forall}(A_{12}) \cup \text{filter}_{\text{lit}}(A_{12}) \quad A_2 = A_1 \cup (A_{12} \setminus A'_{12}) \quad W_2 = W_1 \cup \text{filter}_{\forall}(A_{12}) \quad E_2^I = E_1^I \triangleleft \text{filter}_{\text{lit}}(A_{12}) \quad E_2^H = E_1^H \triangleleft (\sharp\alpha : \vec{r})}{\langle W_1, A_1, E_1 \rangle \longrightarrow \langle W_2, A_2, E_2 \rangle} \text{(INST)}$$

where the overloaded operators filter_\forall and $\text{filter}_{\text{lit}}$ select quantifiers and simple literals, respectively, from any provided set of extended literals, or from unit clauses of any provided set of extended clauses; the judgement $W, E^1 \Vdash_{\text{sat}} C$ holds if: for some disjunct ϕ_i of C , either ϕ_i is a tagged quantifier from W , or ϕ_i is a simple literal that E^1 knows.

Our state transition relation \longrightarrow consists of case-splitting steps, steps that deduce the inconsistent state, steps that deduce the saturated state, and quantifier instantiation steps, corresponding to the rules (SPLIT), (BOT), (SAT) and (INST) respectively.

We allow a case-splitting transition to non-deterministically select *any* non-empty subset of the disjuncts in the *unsatisfied* current clauses—those that have not yet been made true in the current state. A case-splitting transition must make progress towards satisfying the clauses. We do not impose restrictions on the order in which unsatisfied current clauses are chosen, nor on the number of disjuncts assumed within a clause, provided that progress is being made.⁸

We model case-splitting as non-deterministic. Recall Example 2, where the clause $\neg \text{member}(t, a) \vee \neg \text{member}(t, \text{diff}(b, a))$ is learnt. Subsequently, the solver can choose to assume either one or both of the disjuncts; generally, it can choose to assume neither disjunct as long as it selects at least one disjunct from some other unsatisfied clause. Here, the disjuncts are ground simple literals (which are added to the E-state); in general, some could be new quantifiers to record.

Our \Vdash_{sat} judgement checks if a provided clause is satisfied (i.e. at least one disjunct is assumed in the current state). If all current clauses are satisfied, and the E-interface is not inconsistent, and there are no enabled instantiations, the (SAT) rule applies and transitions to the saturated state (\diamond). Conversely, if the current E-interface is inconsistent, the (BOT) rule transitions to the inconsistent state (\perp); if there are enabled instantiations, the (INST) rule applies.

The instantiation rule (INST) relies on the \vdash_{match} judgement to select an instantiation enabled by E-matching rules. The effect of an instantiation transition involves adding quantifiers and simple literals occurring as unit clauses in the quantifier body to the current quantifiers W_1 and E-interface E_1^1 , respectively; any remaining non-unit clauses are added to the current clauses A_1 . Finally, the E-history E_1^H is updated to record this instantiation.

In practice, common SMT solvers such as *cvc5* [2] perform quantifier instantiation both (1) up-front and (2) in phases interleaved with other solver steps. In particular, the latter is essential for many applications: most quantifier instantiations lead to e.g. clauses requiring context-aware case-splitting via DPLL/CDCL. Our model effectively captures both processes through its unrestricted interleavings of quantifier instantiation and case-splitting steps.

In retrospect, Sects. 3.2 to 3.5 have tackled design challenges #1 and #2 (cf. Sect. 2.2). We address #3 and #4 in the next two subsections, respectively.

⁸ Our model allows simulating efficient propagation-based restrictions of case-splitting, but does not require it; restricting to this case would be possible if needed.

3.6 Nested Quantifiers

Example 3 demonstrates that instantiating outer quantifiers in nested structures of quantifiers can introduce new quantifiers on the fly. To effectively argue for termination regarding these instantiations (as will be discussed in Sect. 4), one must be able to identify and predict these dynamically introduced quantifiers. To facilitate this, we employ a tagging system that is capable of handling nested structures (cf. App. A of the TR for details). Each quantifier in an axiomatisation is labelled with a distinct tag. The tag for any non-nested quantifier (including the outermost quantifier in any nested structure of quantifiers) is not parameterised. A nested quantifier has its tag parameterised by all of its outer-quantified variables. Instantiating an outer quantifier produces a copy of the quantifier body in which (among other changes) tags of all inner quantifiers that are parameterised by this outer-quantifier are updated to reflect this instantiation. In Example 3, we label the outer and inner quantifiers with tags $\#union\text{-}elim$ and $\#union\text{-}elim(s_1, s_2)$, respectively. Instantiating the outer quantifier with $s_1 \mapsto a$ and $s_2 \mapsto b$ introduces a copy of the quantifier body in which the inner quantifier is tagged with $\#union\text{-}elim(a, b)$.

To further mitigate redundancy in quantifier instantiation, our semantics supports two additional optimisations. First, a quantifier is only permitted to join the current quantifiers W if its tag is known to be *distinct* from the tags of existing quantifiers in W , *modulo equivalence on the parameters of the tags*, as assessed in the current E-interface. This criterion prevents adding redundant quantifiers into W . Second, the relation of history-enabled E-matches \Vdash_{hist} leverages the current E-interface to verify the uniqueness of tags—once again, modulo equivalence on tag parameters—before enabling an E-match. An E-match is enabled only if no quantifier with an equivalent tag has been instantiated with an equivalent match previously. (cf. App. A of the TR for related definitions.)

3.7 Theory-Specific Reasoning

Although our rules do not yet account for (interpreted) theory reasoning (as performed by theory solvers in a typical SMT solver design), our small-step semantics is intentionally chosen to easily accommodate future extensions: “hot-plugging” new kinds of primitive transitions is straightforward, and will not disturb the existing formal rules (e.g. for quantifier instantiations or case-splitting). Similarly to our E-interfaces for abstracting of E-graph details, we plan to do this in a way which abstracts over the *effects* of theory deduction steps, without exposing the solver-specific internals. For example, we can add deduction steps which extend the E-interface with new terms and/or (dis)equalities, based on a valid deduction within, say, an integer theory.

Just as for quantifier instantiations, it may be necessary for some applications to guarantee that theory reasoning is performed under some fairness conditions (e.g. that inconsistencies detectable by a theory solver are not infinitely postponed). Imposing custom fairness constraints on the traces of our semantics for specific examples can be achieved in a standard way for small-step semantics.

While it is clear that extensions to theory solving will be straightforward, we choose the case study for this paper to be a complex and practically-relevant axiomatisation which nonetheless does not rely on external theory solvers.

4 Proving Instantiation Termination for E-matching

We now apply our model to prove instantiation termination for a practical E-matching-based axiomatisation. First, we briefly present our set theory axiomatisation, adapted from Dafny and Viper. We then demonstrate our methodology for constructing instantiation termination proofs using our model.

4.1 Axiomatisation for Set Theory

To assess our formal model, we tackle formal proofs of instantiation termination for axiomatisations currently employed by state-of-the-art verification tools, specifically targeting set theory in this paper. Set theory, despite the known challenges associated with its quantifier instantiation, is extensively used in verifiers.

Drawing from the axioms used by Dafny [18] and Viper [27], we aim to construct an axiomatisation that (1) faithfully models the core of set theory, (2) supports various encodings of set theory used by verifiers, and (3) strives to maintain a balance on triggers to ensure instantiation termination without harming instantiation completeness.

Our axiomatisation involves 12 uninterpreted functions, representing a wider range of set operations than the counterparts in Dafny and Viper. Cardinality operators are, however, removed due to their dependency on external linear arithmetic solvers (cf. Sect. 3.7 for explanation). Refer to App. C.1 and C.2 of the TR for a full presentation of our axiomatisation and comparison with theirs.

Dafny and Viper typically use complex “iff” formulas to define set operations, restricting trigger flexibility as they must apply in both directions of the “iff”. Inspired by proof systems for formal logic, we redefine set operations using analogues of introduction and elimination axioms, introducing independent triggers for each implication direction and thereby enhancing trigger flexibility.

Example 5. Below is our elimination rule for set union, named (union-elim), allowing more alternative triggers than the counterparts from Dafny and Viper.

$$\begin{array}{l} \forall s_1, s_2, x. [member(x, union(s_1, s_2))] \\ [union(s_1, s_2), member(x, s_1)] [union(s_1, s_2), member(x, s_2)] \\ member(x, union(s_1, s_2)) \rightarrow member(x, s_1) \vee member(x, s_2) \end{array}$$

Our axiomatisation overall has more permissive triggers, which provides more flexibility for instantiation, but also increases the risk of non-termination. That instantiation termination holds for our axiomatisation means that Dafny and Viper’s more restrictive triggers are not necessary to ensure termination.

4.2 Progress Measure

To prove *instantiation termination* for an axiomatisation, it suffices to prove that querying *any* set of ground literals on the axiomatisation cannot lead to an infinite trace in our formal semantics. The proof argument is parametric with respect to the ground literals in the initial state.⁹ Drawing inspiration from program reasoning [7, 26], we identify a suitable measure on solver states and then establish its decrease at appropriate steps in a well-founded manner.

This method leverages the specific features of the axioms under consideration. We analyse our set theory axioms and classify them by two criteria: (1) whether instantiating the axiom would potentially generate new quantifiers or new equivalence classes of terms, i.e. new terms modulo equalities, and (2) whether the axiom contains nested quantifiers.

Non-generative Quantifiers. We call a quantifier *non-generative* if its instantiations yield neither new quantifiers nor new equivalence classes of terms. The majority of our set theory axioms are non-generative.

For instance, the (union-elim) axiom from Example 5, when instantiated with $s_1 \mapsto a$, $s_2 \mapsto b$ and $x \mapsto t$, yields $\neg member(t, union(a, b)) \vee member(t, a) \vee member(t, b)$, without the potential (via case-splitting) to introduce new quantifiers or new equivalence classes of terms. The absence of new terms is because all of t , a , b and $union(a, b)$ are subterms of the matched trigger and hence known. *Bool*-sorted terms never add new equivalence classes (cf. Definition 3).

Instantiating a non-generative quantifier reduces the amount of enabled E-matches by at least one since, on the one hand, history-enabled E-matches prevent instantiating the same quantifier with equivalent matches; on the other hand, instantiating a non-generative quantifier does not introduce new quantifiers or equivalence classes, thereby not expanding the match pool. This suggests:

Idea 1. *Define the progress measure to be about the amount of enabled E-matches.*

Generative Quantifiers. A quantifier is *generative* if its instantiations may introduce new quantifiers or new equivalence classes of terms. Among our set theory axioms *without nested quantifiers*, four are generative, with each potentially creating new applications of Skolem functions upon instantiation.

For instance, the following (subset-intro) axiom, when instantiated, may create a new term $Sk_{ss}(s_1, s_2)$ for some sets s_1 and s_2 :

$$\forall s_1, s_2. [subset(s_1, s_2)] (subset(s_1, s_2) \vee member(Sk_{ss}(s_1, s_2), s_1)) \wedge (subset(s_1, s_2) \vee \neg member(Sk_{ss}(s_1, s_2), s_2))$$

⁹ In fact, it would be straightforward to generalise the termination proof argument, including the termination theorem, to the ground *clauses* in the initial state.

Similarly, axioms for introducing extensional equality on sets, set disjointness, and set emptiness—namely (equal-sets-intro), (disjoint-intro), and (isEmpty-intro-1), respectively—can each produce new applications of Skolem functions: $Sk_{eq}(s_1, s_2)$, $Sk_{dj}(s_1, s_2)$, and $Sk_{ie}(s)$, respectively (cf. App. C.1 of the TR).

Generative quantifiers, by introducing new equivalence classes of terms, may expand the pool of E-matches, including those enabled. We thereby suggest:

Idea 2. *Predict new equivalence classes of terms introduced by instantiating generative quantifiers; incorporate these forecasts to estimate enabled E-matches.*

Set theory axioms *with nested quantifiers* are all generative because their instantiations can potentially create new quantifiers. Such axioms include (subset-elim) from Example 3, and axioms (disjoint-elim) and (isEmpty-elim-1) for eliminating set disjointness and emptiness, respectively (cf. App. C.1 of the TR).

Instantiating these three axioms does not introduce new equivalence classes of ground terms. However, since they contain nested quantifiers, their instantiations can create new quantifiers—each with its own set of enabled E-matches, effectively raising the total amount of enabled E-matches. We therefore propose:

Idea 3. *Incorporate predicted effects from instantiating generative quantifiers with nested quantifier structures to refine estimates of enabled E-matches.*

In practice, provided that these ideas are respected, one can often define simpler termination measures via *over-approximations* of these candidate instantiations (provided this over-approximation remains finite and decreasing).

Formalising a Practical Progress Measure. A basis of an E-interface is a representation of the known equivalence classes. We define its overapproximation to include potential new equivalence classes introduced by generative quantifiers.

Definition 9 (Overapproximation of Basis for Set Theory). *Suppose B is a basis of an E-interface. The functions $O_1(B)$ and $O_2(B)$ denote overapproximations for the $Set(T)$ -sorted and T -sorted elements within basis B , respectively, to accommodate new expected equivalence classes of terms.*

$$\begin{aligned}
 O_1(B) &= \text{filter}_{Set(T)}(B) \\
 O_2(B) &= \text{filter}_T(B) \cup \widehat{Sk}_{ss}(O_1(B), O_1(B)) \cup \widehat{Sk}_{eq}(O_1(B), O_1(B)) \\
 &\quad \cup \widehat{Sk}_{dj}(O_1(B), O_1(B)) \cup \widehat{Sk}_{ie}(O_1(B))
 \end{aligned}$$

Here $\text{filter}_{Set(T)}$ and filter_T take a basis and select its $Set(T)$ -sorted and T -sorted elements, respectively; each \widehat{Sk} is lifted from the corresponding Sk to support sets.

The potential new terms introduced by generative quantifiers are all T -sorted Skolem terms. Thus predictions are solely performed by $O_2(B)$, not by $O_1(B)$.

Note that the results of these two overapproximations are guaranteed to be finite. E-interface bases always remain finite: elements are added (at most) for

the new terms introduced in a step. Since our construction filters and e.g. maps Skolem functions over these finite sets, its results are finite. Leveraging this overapproximation of equivalence classes, we estimate enabled E-matches.

Definition 10 (Overestimation of Enabled E-matches for Set Theory).

Consider an arbitrary state $s = \langle W, A, E \rangle$. Let B be a basis of the E-interface E^I . Define an overestimation of the enabled E-matches for s from B as follows:

$$P(\langle W, A, E \rangle, B) = \{ \dots p_{\# \tau_i}, \dots, p_{\# \tau_j(\vec{\tau})}, \dots \}$$

where $p_{\# \tau_i}$ and $p_{\# \tau_j(\vec{\tau})}$ each denote a set of tuples that overapproximate the enabled E-matches from the basis B to the quantifiers with tags $\# \tau_i$ and $\# \tau_j(\vec{\tau})$, respectively; each tag $\# \tau_i$ identifies an original quantifier from W , and each $\# \tau_j(\vec{\tau})$ identifies a quantifier introduced by instantiating an original quantifier $\# \tau_j$ from W with terms $\vec{\tau}$ from approximations $O_1(B)$ or $O_2(B)$. Original quantifiers from W are those from the axiomatisation, not those introduced at run-time.

To clarify, examples for each category are presented as follows; the remaining quantifiers shall adhere to the same pattern.

- An (original) non-generative quantifier:

$$p_{\# \text{union-elim}} = \{ (s_1, s_2, x) \mid s_1, s_2 \in O_1(B), x \in O_2(B), \\ E \Vdash_{\text{hist}} (\# \text{union-elim} : (s_1, s_2, x)) \}$$

- An (original) generative quantifier without nested quantifiers:

$$p_{\# \text{subset-intro}} = \{ (s_1, s_2) \mid s_1, s_2 \in O_1(B) ; E \Vdash_{\text{hist}} (\# \text{subset-intro} : (s_1, s_2)) \}$$

- An (original) generative quantifier with nested quantifiers:

$$p_{\# \text{subset-elim}} = \{ (s_1, s_2) \mid s_1, s_2 \in O_1(B) ; E \Vdash_{\text{hist}} (\# \text{subset-elim} : (s_1, s_2)) \}$$

- A quantifier introduced by instantiating an (original) generative quantifier:

$$p_{\# \text{subset-elim}(a,b)} = \{ x \mid x \in O_2(B) ; E \Vdash_{\text{hist}} (\# \text{subset-elim}(a,b) : x) \}$$

where $a, b \in O_1(B)$.

We define a progress measure for our set theory axiomatisation. The first and foremost ingredient of our progress measure is an overestimation on the amount of enabled E-matches. We anticipate that this overestimation strictly descends after each instantiation step and does not ascend after each case-splitting step. The second ingredient is the amount of unsatisfied current clauses, which we expect to descend by at least one after each case-splitting step. The result of the progress measure is a lexicographically ordered pair of the above two ingredients.

Definition 11 (Progress Measure for Set Theory). We define the progress measure $M : \text{STATE} \rightarrow (\mathbb{N} \cup \{-1\})^2$, as follows, where $\|\cdot\|$ denotes cardinality.

$$M(s) = \begin{cases} \left(\sum_{p \in P(\langle W, A, E \rangle, B)} \|p\|, \|\{C \in A \mid W, E^I \not\models_{\text{sat}} C\}\| \right) & \text{if } s = \langle W, A, E \rangle \\ & \text{and } B \text{ is a basis for } E^I \\ (-1, -1) & \text{if } s = \perp \text{ or } \diamond \end{cases}$$

Inconsistent or saturated states are assigned (the smallest) measures $(-1, -1)$. The order on $(\mathbb{N} \cup \{-1\})^2$ is the natural extension of that on \mathbb{N} .

4.3 Invariants and Termination Theorem

Drawing on program reasoning, we anticipate classical techniques such as induction variants can be employed to termination proofs. We maintain two kinds of induction variants: general-purpose and problem-specific invariants.

General-purpose invariants uphold the integrity of our formal semantics, remaining valid across all applications. For example, the E-history E^H of an arbitrary state $s = \langle W, A, E \rangle$ must be up to date w.r.t. the current quantifiers W and E-interface E^I . That is, for every pair $(\sharp\tau : \vec{r})$ from E^H , there exists a quantifier $\forall \vec{x}. [\vec{T}]A$ from W whose tag is $\sharp\tau$, the dimension of \vec{x} is equal to that of \vec{r} , $E^I \Vdash_{\text{kn}} \vec{r}$, and $E^I \Vdash_{\text{kn}} \vec{t} [\vec{r}/\vec{x}]$ for some trigger set \vec{t} from $[\vec{T}]$. (cf. App. A of the TR for more invariants.)

Problem-specific invariants are tailored to the distinct features of each problem, focusing on properties of solver states reachable from specified initial states, and tracing the origins of terms in intermediate states. For example, consider an arbitrary intermediate state $\langle W, A, E \rangle$: for each extended clause in A of the form $\neg \text{member}(t, \text{union}(a, b)) \vee \text{member}(t, a) \vee \text{member}(t, b)$, $(\sharp \text{union-elim} : (a, b, t)) \in E^H$ holds; the tag being for the axiom (union-elim) discussed in Example 5. This invariant concerns the origins of the extended clauses in the current clauses A . Case-splitting on a current clause (e.g. the one above) may seem to introduce a new term, but this invariant indicates that this term is not new—it is equal to a known term that triggered a prior instantiation, as tracked by the E-history E^H . This ensures a traceable lineage for each clause, linking it back to a specific quantifier in the E-history. (cf. App. B of the TR for more invariants.)

We finally define the instantiation termination theorem for our set theory axiomatisation, proven by induction on traces leveraging both general-purpose and set-theory-specific invariants. Note that termination is proved against an *arbitrary* set of ground literals—this works because our progress measure and invariants are defined parametrically with the current state. Given these right invariants and termination measure, the proof is straightforward (cf. App. B of the TR). This theorem guarantees the absence of matching loops in this axiomatisation; practitioners of this axiomatisation hence can confidently seek terminating answers to ground theory queries.

Theorem 1 (Instantiation Termination for Set Theory). *Suppose L is an arbitrary set of ground literals. The initial state is $s_0 = \langle W_0, A_0, E_0 \rangle$, where W_0 is our axiomatisation for set theory with tags, $A_0 = \emptyset$, $E_0^I = \emptyset \triangleleft L$, and $E_0^H = \emptyset$. Any sequence of transitions from the initial state s_0 , where \longrightarrow defined in Sect. 3.5 represents the transition relation, has a finite length.*

5 Related Work

For the purpose of program verification, where SMT solvers are used to prove unsatisfiability, E-matching is widely used to handle quantifiers. The idea of E-matching dates back to Nelson [25], which was first put into practice in Simplify [8]. Since then, efficient handling of E-matching-based quantifier instantiation has been studied by, e.g. de Moura and Bjørner [21] for Z3, Ge et al. [14] for CVC3, Bansal et al. [1] for Z3 and CVC4, and Moskal et al. [20] for Fx7. When satisfiable results and their models are of interest, model-based quantifier instantiation (MBQI) [15] can be used to handle quantifiers.

Dross et al. [9–11] formally define and reason about instantiation termination in a similar context. They define a novel *logic* with first-class triggers, introduce *instantiation trees* as algebraic objects to help define termination, and provide an ingenious technique for showing, for their implementation in Alt-Ergo, that finding a *single* finite instantiation tree is sufficient for termination.

Despite being a powerful tool for numerous deep meta-theoretic results [9], we believe that *applying* a formal inductive construction of instantiation trees for larger examples would be complex in practice: existing examples focus instead on bounds for the sets of terms ever generatable by a solver run. These arguments closely relate to our inductive termination proofs over traces. Our work enables detailed formal proofs based directly on such familiar notions from program reasoning, including inductive invariants and well-founded measures.

The approach of this prior work also requires restrictions on solver behaviour, including *fairness* of quantifier instantiation, and *eager* application of theory deductions (via entailments in their custom logic)¹⁰. Our operational model and termination proofs do not require or build in such assumptions. Still, *restricting* our traces (e.g. with fairness constraints) would be simple to do if desired for specific applications. Our weak assumptions make our approach (extended with appropriate theory deduction steps) applicable to SMT solvers broadly; solvers such as Z3 [22] and cvc5 [2] commonly interleave theory reasoning and quantifier instantiation in (bounded or exhaustive) rounds of multiple steps.

The Axiom Profiler [4] leverages Z3 log files to provide comprehensive support for analysing quantifier instantiations. The tool focuses on helping users effectively understand and debug problematic solver runs, rather than proving their absence. It was validated by empirical evidence rather than formal proofs.

Existing works on the termination of SMT transition systems [3, 5, 6, 23] demonstrate that divergence is prevented by ensuring all new terms derive from a finite basis. In contrast, in our work, a finite basis does not imply termination—the basis can grow. At a high level these works prove that certain solver aspects always terminate. However, E-matching cannot have this property; instead it places the onus on the author of an axiomatisation to achieve termination through careful selection of axioms and triggers, motivating a user-facing model.

¹⁰ We explain how to simply add theory steps to our operational model in Sect. 3.7.

6 Conclusion and Future Work

We have shown a novel model for E-matching as widely employed in SMT solvers, abstracting over solver details while enabling detailed and formal proofs of instantiation termination. Our model has been shown to apply directly and rigorously to the kinds of axiomatisations used in practical verification tools.

In future work, we would like to explore axiomatisations that rely on more-restricted characteristics of a solver, such as fairness of instantiation selection or theory reasoning steps. Similarly to our E-interfaces, we will investigate suitable abstractions over theory solver interactions incorporated into a proof search.

While instantiation termination is a much sought-after property, the complementary problem of guaranteed instantiation completeness is a natural next target to investigate with our novel operational model, which may require us to also explore various fairness restrictions of our model's transition relation.

Acknowledgments. We thank the anonymous reviewers, Mark R. Greenstreet and Yanze Li for their detailed and constructive suggestions. We are very grateful to Claire Dross for putting generous time and energy into thoughtful feedback for us. This work has been partly funded by NSERC Discovery Grants held by Garcia and Summers.

References

1. Bansal, K., Reynolds, A., King, T., Barrett, C., Wies, T.: Deciding local theory extensions via e-matching. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 87–105. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_6
2. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006). https://doi.org/10.1007/11916277_35
4. Becker, N., Müller, P., Summers, A.J.: The axiom profiler: understanding and debugging SMT quantifier instantiations. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 99–116. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_6
5. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Conflict-driven satisfiability for theory combination: transition system and completeness. *J. Autom. Reason.* **64**(3), 579–609 (2020). <https://doi.org/10.1007/s10817-018-09510-y>
6. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Conflict-driven satisfiability for theory combination: lemmas, modules, and proofs. *J. Autom. Reason.* **66**(1), 1–49 (2022). <https://doi.org/10.1007/s10817-021-09606-y>
7. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. *Commun. ACM* **54**(5), 88–98 (2011). <https://doi.org/10.1145/1941487.1941509>
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *J. ACM* **52**(3), 365–473 (2005). <https://doi.org/10.1145/1066100.1066102>

9. Dross, C.: Generic decision procedures for axiomatic first-order theories. Ph.D. thesis, Université Paris Sud - Paris XI (2014). <https://tel.archives-ouvertes.fr/tel-01002190>
10. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Reasoning with triggers. In: Fontaine, P., Goel, A. (eds.) SMT 2012. EPiC Series in Computing, vol. 20, pp. 22–31. EasyChair (2013). <https://doi.org/10.29007/3C1N>
11. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Adding decision procedures to SMT solvers using axioms with triggers. *J. Autom. Reason.* **56**(4), 387–457 (2016). <https://doi.org/10.1007/s10817-015-9352-2>
12. Filiâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
13. Ge, R., Garcia, R., Summers, A.J.: A formal model to prove instantiation termination for E-matching-based axiomatisations (extended version). Technical report. [arXiv:2404.18007](https://arxiv.org/abs/2404.18007) (2024). <https://doi.org/10.48550/arXiv.2404.18007>
14. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 167–182. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_12
15. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
16. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 361–381. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_20
17. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
18. Microsoft: Set Axiomatisation (2014). <https://github.com/dafny-lang/dafny/blob/master/Source/DafnyCore/DafnyPrelude.bpl>. Accessed 05 Feb 2024
19. Moskal, M.: Programming with triggers. In: SMT 2009, pp. 20–29. Association for Computing Machinery, New York (2009). <https://doi.org/10.1145/1670412.1670416>
20. Moskal, M., Łopuszański, J., Kiniiry, J.R.: E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.* **198**(2), 19–35 (2008). <https://doi.org/10.1016/j.entcs.2008.04.078>
21. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
22. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
23. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_1
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2

25. Nelson, C.G.: Techniques for program verification. Technical report CSL-81-10, Xerox Palo Alto Research Center (1981)
26. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69. University Mathematical Laboratory, Cambridge, UK (1949)
27. Viper Project Team: Set Axiomatisation (2021). https://github.com/viperproject/carbon/blob/master/src/main/scala/viper/carbon/modules/impls/sequence_axioms/SetAxiomatization.scala. Accessed 05 Feb 2024

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

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





Fast and Verified UNSAT Certificate Checking

Peter Lammich  

University of Twente, Enschede, Netherlands
p.lammich@utwente.nl

Abstract. We describe a formally verified checker for unsatisfiability certificates in the LRAT format, which can be run in parallel with the SAT solver, processing the certificate while it is being produced. It is implemented time and memory efficiently, thus increasing the trust in the SAT solver at low additional cost.

The verification is done w.r.t. a grammar of the DIMACS format and a semantics of CNF formulas, down to the LLVM code of the checker. In this paper, we report on the checker and its design process using the Isabelle-LLVM stepwise refinement approach.

Keywords: UNSAT certificates · LRAT · Isabelle-LLVM · Verified Software

1 Introduction

SAT solvers are highly complex and highly optimized programs, which are used to verify critical properties of other systems. To increase the trust in them, SAT solvers produce certificates that can be independently checked by formally verified checkers [5, 9, 10, 16, 23, 34, 35]. Here, the focus is on certificates for unsatisfiability, as certificates for satisfiability are (considered) trivial.

Typically, certificate checking proceeds in two phases: An unverified *elaborator* adds additional information to the certificate produced by the SAT solver, and then a formally verified *checker* checks the elaborated certificate against the original formula. This approach moves some complicated and computationally expensive tasks into the unverified elaborator, making checking of the elaborated certificate simpler and less expensive.

However, the elaborator has to recompute information which is, in principle, known to the solver, and elaboration typically takes as long as solving. More recent techniques accelerate elaboration by including this information into the certificate [2]. The most recent development are solvers that directly produce elaborated certificates [29]. This allows for *streaming* the certificates from the solver into the checker: solving and checking are done in parallel, and the potentially large certificates need not be stored on disk. When implemented appropriately, the memory footprint of the checker is similar to that of the solver.

There are different formats for elaborated unsatisfiability certificates, such as PB [4] and GRAT [23]. The de-facto standard is the LRUP format [10], and its backwards compatible generalizations LRAT [9] and LPR [35]. These correspond to the non-elaborated DRUP [17], DRAT [36], and DPR [35] formats. With an exception in 2023, LRUP is sufficient for all top performing solvers in the SAT competitions of the last years [29].

In this paper, we present a formally verified checker that can stream LRUP certificates. We benchmark our tool on the CaDiCaL solver [29], where it only causes a minimal additional computation overhead, and has a memory usage similar to that of the solver. Our checker is as fast as the highly optimized unverified *lrat-trim* checker [29], and at least one order of magnitude faster than any other verified checker we know of. Using the Isabelle Refinement Framework [22], our checker is verified down to the LLVM intermediate representation [26] of its code, and against a formal grammar of the DIMACS CNF format, which is the standard for representing CNF formulas [32]. To the best of our knowledge, our checker is the first that comes with a verified parser. Our tool and benchmark data is available at https://github.com/lammich/lrat_isa.

In the rest of this paper, we describe our formal specification (Sect. 2), the abstract certificate checking algorithm (Sect. 3), and its implementation (Sect. 4). We then report on our benchmark results (Sect. 5). Finally we conclude the paper and discuss related and future work (Sect. 6).

2 Specification

We prove soundness of our checker, i.e., it accepts a string only if it is a representation of an unsatisfiable formula in DIMACS CNF format¹. In this section we present the formalization of this specification.

2.1 Conjunctive Normal Form

Throughout this paper, we will use some simplified Isabelle/HOL notation, and explain unusual notations where they first occur. For definitions we use \equiv . Data types are written in prefix notation, e.g., *lit set* for a set of literals. Function application is denoted as $f x_1 \dots x_n$.

The following is the abstract syntax and semantics of CNF, taken from the GRAT tool [23] and slightly adapted to our needs:

```
typedef var  $\equiv$  {v::nat. v  $\neq$  0}
lit  $\equiv$  Pos var | Neg var    clause  $\equiv$  lit set    cnf  $\equiv$  clause set

valuation  $\equiv$  var  $\Rightarrow$  bool
sem_lit :: lit  $\Rightarrow$  valuation  $\Rightarrow$  bool
```

¹ Note that proving completeness is less interesting: even if we show that our checker accepts all valid certificates, the elaborator or solver may still fail to produce one. We verify completeness empirically on a large set of benchmarks.

$$\text{sem_lit } (\text{Pos } v) \sigma \equiv \sigma v \qquad \text{sem_lit } (\text{Neg } v) \sigma \equiv \neg \sigma v$$

$$\begin{aligned} \text{sem_cnf} &:: \text{cnf} \Rightarrow \text{valuation} \Rightarrow \text{bool} \\ \text{sem_cnf } F \sigma &\equiv \forall C \in F. \exists l \in C. \text{sem_lit } l \sigma \end{aligned}$$

$$\text{sat} :: \text{cnf} \Rightarrow \text{bool} \qquad \text{sat } F \equiv \exists \sigma. \text{sem_cnf } F \sigma$$

A *variable* is a positive natural number, a *literal* is a positive or negative variable, a *clause* is a set of literals, and a *cnf-formula* is a set of clauses. A *valuation* assigns truth values to variables. For a valuation σ , the *semantics* assigns truth values to literals (*sem_lit*) and formulas (*sem_cnf*): a positive literal is true iff its variable is true, and a negative literal is true iff its variable is false. A formula is true iff every clause contains a true literal, and it is *satisfiable* if there is a valuation for which it is true.

2.2 Specification of the DIMACS CNF Format

DIMACS CNF is the de-facto standard format for representing CNF formulas. Figure 1 displays an example: the file can start with optional comment lines, indicated by a heading ‘c’. After the comments, there is a header of the form **p cnf n m**, where n is the maximum variable, and m is the number of clauses. Then the clauses follow, encoded as zero-terminated sequences of integers, where a positive integer represents a positive literal, and a negative integer represents a negative literal. We need to specify how a word in DIMACS format corresponds to a formula. While a language is a set of words, we use a relation between words and corresponding abstract syntax. By slight abuse of naming, we call such relations *grammars*. We shallowly embed regular grammars into Isabelle HOL’s logic:

```
c start with comments
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Fig. 1. Example formula in DIMACS CNF

$$\begin{aligned} ('a, 'r) \text{ gM} &\equiv ('a \text{ list} \times 'r) \text{ set} \\ \text{return } x &\equiv \{ ([], x) \} \qquad \langle C \rangle \equiv \{ ([c], c) \mid c \in C \} \\ \text{bind } g f &\equiv \{ (w_1 @ w_2, r) \mid \exists x. (w_1, x) \in g \wedge (w_2, r) \in f x \} \end{aligned}$$

Here, $(w, r) \in g$ means that the grammar g relates the word w to the result r . The empty relation $\{\}$ corresponds to the empty language. The relation **return** x relates the empty word to the result x . It corresponds to the language $\{[]\}$ of only the empty word. The relation $\langle C \rangle$ relates single-character words to the corresponding character from the set C . Finally, the relation **bind** $g f$ relates a word $w_1 w_2$ to a result r , if g relates w_1 to some intermediate result x , and $f x$ relates w_2 to r . This corresponds to concatenation of languages.

The type gM is a monad, and we use the usual shortcut notation for **bind**:

$$x \leftarrow g; f x \equiv \text{bind } g (\lambda x. f x) \qquad g_1; g_2 \equiv \text{bind } g_1 (\lambda _ . g_2)$$

We also define shortcuts to apply a function to the result of a monad, to lift a binary function into a monad, and to concatenate two grammars, ignoring the result of the latter:

$$\begin{aligned} a \langle \& \rangle f &\equiv x \leftarrow a; \text{ return } (f x) \\ \text{lift2 } f \ a \ b &\equiv x \leftarrow a; y \leftarrow b; \text{ return } (f x y) \\ a \ll b &\equiv r \leftarrow a; b; \text{ return } r \end{aligned}$$

We then define the relational versions of the power function and the Kleene star:

$$\begin{aligned} g_pow \ g \ 0 &\equiv \text{ return } [] & g_pow \ g \ (n+1) &\equiv \text{lift2 } (\#) \ g \ (g_pow \ g \ n) \\ g^* &\equiv \bigcup_{n::\text{nat.}} g_pow \ g \ n \end{aligned}$$

where $x\#xs$ prepends the element x to the list xs . That is, $g_pow \ g \ n$ and g^* relate the input to lists, the elements being the results produced by g . We also define $g^?$ $\equiv (g \langle \& \rangle \text{ Some}) \cup (\text{ return } \text{ None})$.

Using the grammar monad, we specify a grammar for the simplified DIMACS format as used by SAT competitions since 2009 [32]. We start with defining sets of ASCII characters:

$$\begin{aligned} \text{whitespace, digits1, digits} &:: 8 \text{ word set} \\ \text{whitespace} &\equiv \{ ' ', '\t', '\n', '\v', '\f', '\r' \} \\ \text{digits1} &\equiv \{ '1', \dots, '9' \} & \text{digits} &\equiv \{ '0', \dots, '9' \} \end{aligned}$$

Here, 8 word is the 8-bit word type from Isabelle's machine word library [3,11]. Note whitespace includes all 6 ASCII whitespace characters. Based on this, we define a grammar $g_dimacs :: (8 \text{ word} \times \text{cnf}) \text{ set}$:

$$\begin{aligned} g_ws &\equiv \langle \text{whitespace} \rangle^*; \text{ return } () & g_ws1 &\equiv \langle \text{whitespace} \rangle; g_ws \\ g_variable &\equiv x \leftarrow \langle \text{digits1} \rangle; xs \leftarrow \langle \text{digits} \rangle^*; \text{ return } (\text{nat_of_str } (x\#xs)) \\ g_literal &\equiv \langle \{ '-' \} \rangle; g_variable \langle \& \rangle \text{ Neg} \cup (g_variable \langle \& \rangle \text{ Pos}) \\ g_clause &\equiv (g_literal \ll g_ws1)^* \langle \& \rangle \text{ set} \ll \langle \{ '0' \} \rangle \\ g_cnf &\equiv (\text{ return } \{ \}) \\ &\cup (c \leftarrow g_clause; cs \leftarrow (g_ws1; g_clause)^*; \text{ return } (\{c\} \cup \text{set } cs)) \\ g_comment &\equiv \langle \{ 'c' \} \rangle; \langle -\{ '\n' \} \rangle^*; \langle \{ '\n' \} \rangle; \text{ return } () \\ g_p_header &\equiv \langle \{ 'p' \} \rangle; \langle -\{ '\n' \} \rangle^*; \langle \{ '\n' \} \rangle; \text{ return } () \\ g_comments &\equiv (g_ws \cup g_comment)^*; \text{ return } () \\ g_dimacs &\equiv g_comments; g_p_header^?; g_ws; g_cnf \ll g_ws \end{aligned}$$

Here, $\text{nat_of_str} :: 8 \text{ word list} \Rightarrow \text{nat}$ converts a string to a natural number, and $\text{set} :: 'a \text{ list} \Rightarrow 'a \text{ set}$ yields the set of elements in a list.

Note that we do not check the contents of the header, which contains auxiliary information for parsing, but does not affect the represented formula. We also accept multiple clauses per line and clauses spanning several lines, as well as extra whitespace anywhere in the file. Many SAT solvers support similar relaxations of the format, and we wanted this flexibility in our tool, too.

As a sanity check, we prove that our grammar is unambiguous, i.e., that it relates the same word to at most one formula:

$$(w, f_1) \in g_dimacs \wedge (w, f_2) \in g_dimacs \implies f_1 = f_2$$

2.3 Correctness Specification

At this point, we can formalize the postcondition for our checker’s specification: $\exists F. (w, F) \in g_dimacs \wedge \neg sat F$ means that the sequence of bytes w is a valid DIMACS CNF representation of an unsatisfiable formula.

3 Certificates for Unsatisfiability

RUP (reverse unit propagation) certificates contain the clauses learned by the solver. The checker justifies that addition of each clause preserves satisfiability. For an unsatisfiable formula, the last learned clause is the empty clause. Adding the empty clause yields an unsatisfiable formula, and, as each clause addition is justified to preserve satisfiability, the original formula is unsatisfiable, too.

Justification is done by *reverse unit propagation* [14]: a clause C can be added to the formula F , if the formula $F \wedge \neg C$ is unsatisfiable, and if this can be shown by generating an empty clause via unit propagation. For RUP, the checker has to implement unit propagation itself, for example with a two-watched-literals data structure [28]. LRUP (linear RUP) certificates annotate each clause addition, with the relevant unit clauses in the order they become unit, and the final conflict clause. This makes the checker simpler and more efficient, as it only needs to check if clauses are unit, rather than find unit clauses.

The certificates also contain clauses deleted by the solver. This allows the checker to also delete those clauses from its data structures, freeing up memory. Note that deleting a clause trivially preserves satisfiability.

The actual LRUP format uses natural numbers to identify clauses, rather than spelling them out whenever they are referenced. The n clauses of the initial formula implicitly get the ids $[1, \dots, n]$. A clause addition has the form $\langle id \rangle \langle literal \rangle^* 0 \langle id \rangle^+ 0$. It consists of the id under which this clause shall be added, a zero-terminated list of the literals of the clause, and a zero terminated list of the unit clauses and the conflict clause to justify the addition. A clause deletion has the form $\langle id \rangle^+ 0$, and consists of a zero terminated list of the ids of the clauses to be deleted. There is an ASCII and a more compact binary encoding for LRUP certificates.

3.1 Abstract Checker

In this section, we present our formalization of the abstract checker algorithm. We start with defining some basic concepts:

$$\begin{aligned}
- & :: \text{lit} \Rightarrow \text{lit} & -\text{Pos } v & \equiv \text{Neg } v & -\text{Neg } v & \equiv \text{Pos } v \\
\text{pan} & \equiv \text{lit} \Rightarrow \text{bool} & \text{consistent } (A::\text{pan}) & \equiv \forall l. \neg (A \text{ l} \wedge A (-l)) \\
\text{sat_wrt } F \ A & \equiv \exists \sigma. \text{sem_cnf } F \ \sigma \wedge (\forall l. A \ \text{l} \Longrightarrow \text{sem_lit } \text{l} \ \sigma) \\
\text{conflict } A \ C & \equiv \forall l \in C. A (-l) \\
\text{is_uot } A \ C \ l & \equiv l \in C \wedge \neg A(-l) \wedge (\forall l' \in C - \{l\}. A(-l')) \\
\text{taut } C & \equiv \exists l. l \in C \wedge -l \in C
\end{aligned}$$

The literal $-l$ is the negation of the literal l . A *partial assignment* (pan) characterizes a set of literals that are assigned (to true). It is *consistent* if it does not assign both a literal and its negation. A formula F is *satisfiable w.r.t.* a partial assignment A ($\text{sat_wrt } F \ A$), if A can be extended to a satisfying valuation; A is in *conflict* with a clause C ($\text{conflict } A \ C$), if the negations of all the clause's literals are assigned. The clause C is *unit or true* w.r.t. A and a literal l ($\text{is_uot } A \ C \ l$), if l is the only literal in C whose negation is not assigned. A clause is a *tautology* ($\text{taut } C$), if it contains both a literal and its negation.

Correctness of a RUP step adding C to F is implied by the following lemmas:

- (1) Let C be a non-tautological clause. Then, the *initial assignment* $\lambda l. -l \in C$, which assigns the negated version of each literal in C , is consistent, and $F \wedge \neg C$ is satisfiable iff F is satisfiable w.r.t. the initial assignment:

$$\neg \text{taut } C \Longrightarrow \text{consistent } (\lambda l. -l \in C) \wedge \text{sat } (F \wedge \neg C) = \text{sat_wrt } F \ (\lambda l. -l \in C)$$

- (2) If the formula contains a unit or true clause, assigning its literal preserves consistency and does not change satisfiability:

$$\begin{aligned}
& \text{consistent } A \wedge C \in F \wedge \text{is_uot } A \ C \ l \\
& \Longrightarrow \text{consistent } (A(l := \text{True})) \wedge \text{sat_wrt } F \ A = \text{sat_wrt } F \ (A(l := \text{True}))
\end{aligned}$$

- (3) If the formula contains a conflict clause, it is unsatisfiable:

$$\text{consistent } A \wedge C \in F \wedge \text{conflict } A \ C \Longrightarrow \neg \text{sat_wrt } F \ A$$

Note that the learned clause cannot be a tautology. While adding tautologies trivially preserves satisfiability, they yield an inconsistent initial assignment. Instead of spending computation time to detect tautologies, we let our checker run with the inconsistent assignment: should it succeed, we add the clause, which is safe.

We formalize the abstract checker as a transition system over the state:

$$\begin{aligned}
\text{checker_state} & \equiv \text{CNF formula} \mid \text{CLS formula clause pan} \\
& \mid \text{PRF formula clause pan} \mid \text{PDN formula clause} \mid \text{UNSAT} \mid \text{FAIL}
\end{aligned}$$

The transition relation \rightarrow is the least relation that satisfies the following rules:

$$\begin{aligned}
(\text{del_clauses}) \quad F' \subseteq F & \Longrightarrow \text{CNF } F \rightarrow \text{CNF } F' \\
(\text{start_clause}) \quad \text{CNF } F \rightarrow \text{CLS } F \ \{\} \ (\lambda _ . \text{False}) \\
(\text{add_lit}) \quad \text{CLS } F \ C \ A \rightarrow \text{CLS } F \ (\{l\} \cup C) \ (A(-l := \text{True})) \\
(\text{start_proof}) \quad \text{CLS } F \ C \ A \rightarrow \text{PRF } F \ C \ A \\
(\text{add_unit}) \quad u \in F \wedge \text{is_uot } A \ u \ C \ ul \\
& \Longrightarrow \text{PRF } F \ C \ A \rightarrow \text{PRF } F \ C \ (A(ul := \text{True}))
\end{aligned}$$

$$\begin{aligned}
& (\text{add_conflict}) \quad uC \in F \wedge \text{conflict } A \ uC \implies \text{PRF } F \ C \ A \rightarrow \text{PDN } F \ C \\
& (\text{finish_proof}) \quad C \neq \{\} \implies \text{PDN } F \ C \rightarrow \text{CNF } (\{C\} \cup F) \\
& (\text{finish_proof_unsat}) \quad \text{PDN } F \ \{\} \rightarrow \text{UNSAT} \\
& (\text{to_fail}) \quad s \rightarrow \text{FAIL}
\end{aligned}$$

The checker starts in state $\text{CNF } F$, with some formula F . To delete clauses (del_clauses), they are removed from F . A clause addition is split into multiple smaller steps: First, we initiate adding a clause by going to state CLS (start_clause). We also maintain a partial assignment, starting with the empty assignment $\lambda_.$ *False*. We then add the literals of the clause, one by one (add_lit). For each added literal l , we assign the negated literal $-l$. When all literals have been added, we start the proof (start_proof) going to state PRF . During the proof, we add unit clauses, assigning the unit literal (add_unit). When we have added enough unit clauses, we add a conflict clause (add_conflict), going to state PDN (proof done). From there, we either go to state UNSAT if we have proved the empty clause ($\text{finish_proof_unsat}$), or back to state CNF with the new clause added to the formula (finish_proof). We can always go to FAIL (to_fail), indicating that the proof failed.

With the above Lemmas 1–3, some bookkeeping that add_lit steps construct the correct initial assignment, and a special case for tautologies, we prove:

Theorem 1 (Soundness of Abstract Checker). *If the abstract checker can reach UNSAT from the initial state $\text{CNF } F$, then the formula F is unsatisfiable: $\text{CNF } F \rightarrow^* \text{UNSAT} \implies \neg \text{sat } F$*

Note that we do not yet model clause identifiers on this level. They will be introduced in a later refinement step.

4 Implementation

We have specified a grammar to relate strings in DIMACS format to formulas, a semantics to define satisfiability of formulas, and an abstract certificate checker. We now refine these to the actual implementation of a certificate checker.

We use the Isabelle Refinement Framework [24], which supports refinement in multiple steps and in a modular fashion. Each step focuses on a different aspect of the algorithm, thus structuring the correctness proof, and making it manageable in the first place. In this section, we first describe the data structures that we use in our implementation, to represent abstract concepts such as literals and clauses (cf. Sect. 2.1). We then describe how we implement the abstract checker algorithm (cd. Sect. 3.1), using these data structures. Finally, we describe how we integrate the checker with the parser, to obtain the actual verified tool.

4.1 Basic Concepts and Data Structures

We use data structures such as arrays, dynamic arrays, and array lists from Isabelle LLVM’s library [22]. For technical reasons, sizes and counters are implemented as non-negative *signed* 64-bit integers, or, equivalently, as unsigned 64-bit integers less than 2^{63} . Formally, *refinement relations* between concrete and

abstract types are used. For example, $size_rel :: (64 \text{ word} \times nat) \text{ set}$ relates non-negative 64-bit signed integers to natural numbers. Similarly, Booleans are implemented by 1-bit words, via the relation $bool1_rel :: (1 \text{ word} \times bool) \text{ set}$.

Clause identifiers are modelled as 64-bit unsigned integers less than $2^{63} - 1$, via the relation $cid_rel :: (64 \text{ word} \times nat) \text{ set}$. This bound allows us to use clause identifiers as indexes into an array whose length is represented by a size.

Literals are first refined to natural numbers via $nlit_rel :: (nat \times lit) \text{ set}$, where a number $n > 1$ represents the variable $\lfloor n/2 \rfloor$, and the literal is negative iff n is odd. The natural numbers are further refined to unsigned 32-bit integers, via $u32_rel :: (32 \text{ word} \times nat) \text{ set}$. When we compose the two refinement relations, we get a relation between 32-bit integers and literals: $ulit_rel \equiv u32_rel \circ nlit_rel$. Using 0 for *None*, we can also refine optional literals to 32-bit integers via the relation $ulito_rel :: (32 \text{ word} \times lit \text{ option}) \text{ set}$. For each operation on the abstract data type, we define a corresponding operation on the concrete data type. For example, we define:

$$\begin{aligned} nlit_neg &:: nat \Rightarrow nat & nlit_neg \ n &\equiv \text{if even } n \text{ then } n+1 \text{ else } n-1 \\ ulit_neg &:: 32 \text{ word} \Rightarrow 32 \text{ word} & ulit_neg \ w &\equiv w \text{ XOR } 1 \end{aligned}$$

We show that the concrete operations refine their abstract counterparts:

$$ulit_neg, nlit_neg :: u32_rel \rightarrow u32_rel \quad nlit_neg, (-) :: nlit_rel \rightarrow nlit_rel$$

Here, $f, g :: R_1 \rightarrow R_2$ is a shorthand notation for $\forall(x,y) \in R_1. (f \ x, g \ y) \in R_2$. Combining these refinement theorems yields $ulit_neg, (-) :: ulit_rel \rightarrow ulit_rel$.

Clauses are implemented as zero-terminated arrays of 32-bit words, via the relation $zcl_assn :: 32 \text{ word ptr} \Rightarrow clause \Rightarrow assn$. As arrays are stored on the heap, this relation is expressed as separation logic assertion (*assn*). By convention, pure refinement relations have the suffix *_rel*, while those that use the heap have the suffix *_assn*.

A *clause database* $cdb \equiv nat \Rightarrow clause \text{ option}$ is a partial function from clause identifiers to clauses. It is implemented by a dynamic array of pointers to clauses $cdbi \equiv 32 \text{ word ptr larray}$, via $cdb_assn :: cdbi \Rightarrow cdb \Rightarrow assn$. The array is indexed by the clause identifier. For clause identifiers not in the database, the array contains a null pointer. Consider the abstract operation $cdb_ins \ cid \ C \ db$ that inserts a clause C with identifier cid into the database db , its concrete version cdb_ins_impl , and the corresponding refinement theorem:

$$\begin{aligned} cdb_ins &:: nat &\Rightarrow clause &\Rightarrow cdb &\Rightarrow cdb \\ cdb_ins_impl &:: 64 \text{ word} &\Rightarrow 32 \text{ word ptr} &\Rightarrow cdbi &\Rightarrow cdbi \\ cdb_ins_impl, cdb_ins &:: cid_rel &\rightarrow zcl_assn^d &\rightarrow cdb_assn^d &\rightarrow cdb_assn \end{aligned}$$

The concrete operation destructively updates the array, thus the abstract cdb parameter does no longer correspond to any concrete value. Also, the ownership of the inserted clause is transferred into the clause database, thus the abstract clause parameter does no longer correspond to any (visible) concrete value. We call those parameters *destroyed*, indicated by a d in the refinement theorem [21].

4.2 Data Structures with Capacity Bounds

Several data structures in our checker use counters. For example, during parsing, the literals of a clause are collected in an array list, which uses a counter for its size. We prove non-overflow of these counters from the bounded size of the CNF file, and a limit on how many literals we can read from the certificate before the checker rejects it². While we elide the details, we note that some abstract data structures have a capacity bound field. This is a *ghost field*, i.e., it is not present in the implementation.

The *clause builder* uses a dynamic array to store the literals of the clause that is currently parsed, and also keeps track of the maximum literal encountered so far. Its abstract type is $cbl \equiv nat \times lit\ list \times nat$. A clause builder $(ml, ls, bnd) :: cbl$ consists of the maximum encountered literal ml , the current list of literals ls , and a (ghost) bound bnd that limits the length of ls . We define a *data type invariant* $cbl_inv :: cbl \Rightarrow bool$ that characterizes valid clause builders (i.e., the bound and maximum literal are consistent with the list of literals). The relation $cbl_assn :: (32\ word \times 32\ word\ array_list) \Rightarrow cbl \Rightarrow assn$ implements clause builders.

A partial assignment (cf. Sect. 3.1) is implemented by an array of bits indexed by the literals, as well as an array list that contains all set literals. This array list allows for efficiently resetting the assignment in between proof steps. We use the type $rpani :: 1\ word\ larray \times 32\ word\ array_list$ for the implementation, and $rpan :: bool\ list \times nat\ list \times nat$ for the functional representation, related by $rpan_assn :: rpani \Rightarrow rpan \Rightarrow assn$. The last field of $rpan$ is a (ghost) capacity bound. The type $rpan$ comes with an invariant $rpan_inv$, and an abstraction function $rpan_a :: rpan \Rightarrow pan$ to the encoded partial assignment.

4.3 Proof Checker Implementation

We implement the abstract checker state (Sect. 3.1) by the following types:

$$\begin{array}{ll} cs_op \equiv bool \times bool \times cdb \times cbl \times rpan & \text{— outside proof (CNF, UNSAT)} \\ cs_bc \equiv bool \times rpan \times cbl \times cdb & \text{— build clause (CLS)} \\ cs_ip \equiv bool \times bool \times rpan \times cbl \times cdb & \text{— inside proof (PRF, PDN)} \end{array}$$

All data structures start with an error flag, which indicates a failed proof (abstract state *FAIL*). Outside a proof, i.e., in abstract states *CNF* and *UNSAT*, the checker state is represented by a tuple $(err, unsat, db, bld, A) :: cs_op$, where *unsat* indicates that the formula has been proved unsatisfiable and *db* is the clause database holding the formula. The builder state *bld* and assignment *A* are unused here, but threaded through such that they can be reused when the next proof begins. When building a clause (abstract state *CLS*), the state is represented as $(err, A, bld, db) :: cs_bc$. Finally, inside a proof (abstract states *PRF* and *PDN*), the state is $(err, confl, A, bld, db) :: cs_ip$. Here, *confl* indicates that a conflict clause has been found.

² The size of the formula plus the number of literals in the certificate cannot exceed 2^{63} . We don't expect this limit to be ever hit in practice.

We define invariants cs_op_inv , cs_bc_inv , cs_ip_inv ; and abstraction functions $cs_op_α$, $cs_bc_α$, $cs_ip_α$ to the abstract checker state. We then show that the functions on the concrete states preserve the invariants and implement the transition relation \rightarrow^* on the corresponding abstract states. For example, the following function handles a proof step, adding a unit or a conflict clause:

$$\begin{aligned} cs_prf_step &:: nat \Rightarrow cs_ip \Rightarrow cs_ip \ nres \\ cs_ip_inv \ cap \ cs &\wedge \ cap > 0 \\ \implies cs_prf_step \ cid \ cs &\leq \mathbf{spec} \ cs'. \ cs_ip_inv \ (cap-1) \ cs' \\ &\wedge \ (cs_ip_α \ cs) \rightarrow^* \ (cs_ip_α \ cs') \end{aligned}$$

Here, $'a \ nres$ is the Isabelle Refinement Framework's type of programs that return a result of type $'a$, and $P \implies c \leq \mathbf{spec} \ r. \ Q \ r$ is a Hoare-triple with pre-condition P , program c , and postcondition Q [24]. That is, if the concrete checker state cs has some capacity left, then the cs_prf_step function preserves the invariant cs_ip_inv and implements the abstract transition relation \rightarrow . The available capacity of the checker state decreases by one.

```

1  check_uot :: rpan  $\Rightarrow$  cdb  $\Rightarrow$  nat
2            $\Rightarrow$  (lit option  $\times$  bool) nres
3  check_uot A cdb cid  $\equiv$  if cid  $\in$  dom cdb then
4    let C = the (cdb cid);
5    assume finite C
6    foreach C ( $\lambda l$  (ul,err).
7      assert rpan_in_bounds (-l) A
8      if A (-l) then return (ul,err)
9      else if ul = None then return (Some l,err)
10     else return (ul,True)
11   ) (None,False)
12 else return (None,True)

```

Fig. 2. Function to check if a clause is unit, true, or conflict.

assigns to ul the first literal that is not false (l. 9). If a second non-false literal is encountered, the error flag is set (l. 10). The function returns the state after the loop, or $(None, True)$ if the clause was invalid (l. 12). Note that we *assume* (l. 5) a finite clause. On the abstract level, we can use this to justify termination of the loop. When implementing the function, we have to prove finiteness, which is trivial, as the clause is stored in an array. Dually, we *assert* (l. 7) that the literals of the clause are in bounds of the assignment. This has to be proved on the abstract level. When implementing, we can use it to justify that the array access for looking up the literal is in bounds. This way, assertions and assumptions are used to pass proof obligations up and down the refinement chain, proving them at the most convenient abstraction level.

The loop in $check_uot$ is the innermost loop of the checker, and special care has been taken to optimize it: while an actual certificate always contains unit clauses, we also allow clauses with one true literal (cf. is_uot in Sect. 3.1). This avoids indexing both $A(l)$ and $A(-l)$ to distinguish between the two cases.

The implementation of cs_prf_step uses a function to check if a clause is unit, true, or a conflict. It is displayed in Fig. 2. It first checks (l. 3) if the clause identifier is valid, and looks it up in the database (l. 4). Then (l. 6), it loops over the literals of the clause, maintaining a state consisting of an optional literal and an error flag (ul, err) . Initially (l. 11), the state is $(None, False)$. The loop

The correctness theorem for *check_uot* is as follows:

$$\begin{aligned} rpan_inv A \wedge cdb\ cid = Some\ C \wedge cdb_vars\ cdb \subseteq rpan_dom\ A &\implies \\ check_uot\ cdb\ cid\ A \leq \mathbf{spec}\ (ul, err). \neg\ err \longrightarrow \mathbf{case}\ ul\ \mathbf{of} & \\ Some\ l \Rightarrow is_uot\ (rpan_alpha\ A)\ C\ l \mid None \Rightarrow \mathbf{conflict}\ (rpan_alpha\ A)\ C & \end{aligned}$$

I.e., if the partial assignment satisfies its invariant, the clause identifier identifies clause C , and the clause database contains only variables within the bounds of the partial assignment, then the function will either return an error, or some literal l such that C is unit or true w.r.t. l , or $None$ and C is a conflict clause.

4.4 A Verified DIMACS Parser

We present the parsing function's signature and correctness theorem. Its implementation is elided due to page limit constraints:

$$\begin{aligned} read_dimacs_cs &:: 8\ word\ list \Rightarrow (cs_op \times nat)\ nres \\ read_dimacs_cs\ str &\leq \mathbf{spec}\ (cs, cap). \exists F. cs_op_inv\ cap\ cs \\ &\wedge (cs_op_alpha\ cs = FAIL \vee (str, F) \in g_dimacs \wedge cs_op_alpha\ cs = CNF\ F) \end{aligned}$$

This function parses a string, and returns a checker state. On a parsing error, the checker state corresponds to the abstract state *FAIL*. Otherwise, it corresponds to *CNF F* for the formula F parsed from the string. The function also returns the capacity left for the certificate after parsing the formula.

4.5 Assembling the Whole Program

Having implemented functions for the proof steps, we combine them with a parser (details elided) for LRAT proofs, resulting in a function that reads an LRAT proof from a buffered reader (*brd_rs*), performs the corresponding transitions on the proof state, and finally checks if the proof state has reached *UNSAT*:

$$main_checker_loop :: cs_op \Rightarrow brd_rs \Rightarrow (bool \times brd_rs)\ nres$$

The certificate checker, displayed in Fig. 3, combines the main checker loop with the DIMACS parser. It takes a string *cnf*, parses it as formula (l. 3), initializes a buffered reader for the certificate stream (l. 5), and runs the main checker loop with that reader (l. 6). From the correctness of the parser (Sect. 4.4), the fact that all proof steps in *main_checker_loop* implement the abstract checker, and the fact that the abstract checker is sound (Theorem 1), we prove:

```

1 read_check_lrat :: 8 word list => bool nres
2 read_check_lrat cnf ≡
3   (cs, cap) ← read_dimacs_cs cnf;
4   if ¬ csop_is_err cs then
5     prf = brd_rs_new cap
6     (res, _) = main_checker_loop cs prf
7     return res
8   else return False

```

Fig. 3. The checker program.

Theorem 2 (Soundness of Functional Checker). *If `read_check_lrat cnf` returns true, then `cnf` is a valid representation of an unsatisfiable formula:*

$$\text{read_check_lrat } cnf \leq \text{spec } r. r \implies \exists F. (cnf, F) \in g_dimacs \wedge \neg \text{sat } F$$

4.6 Refinement to LLVM Code

In Sect. 4.1 and Sect. 4.2 we have indicated how we implement the basic data structures of our checker. Then, we have mostly presented functional code. Given implementations of the data structures, refining this functional code to imperative code is mostly straightforward. Actually, much of this process can be automated by the Sepref tool [22], which we use to generate implementations for each data structure and algorithm. For example, for the function `check_uot` (cf. Sect. 4.3):

```
sepref_def check_uot_impl is
  check_uot :: rpan_assn → cdb_assn → cid_rel → ulito_rel × bool1_rel
  unfolding check_uot_def by sepref
```

This generates the function `check_uot_impl` and proves the refinement theorem:

```
check_uot_impl, check_uot
  :: rpan_assn → cdb_assn → cid_rel → ulito_rel × bool1_rel
```

To read the certificate, we use an external C function based on `fread`:

```
size_t fread_from_certificate(void *p, size_t n) {
  if (!cert_file) return 0;
  return fread(p, 1, n, cert_file); }
```

Inside Isabelle, this function is specified by:

```
htriple (arr_assn xsi xs ★ size_rel ni n ★ n ≤ length xs)
  fread_from_certificate xsi ni
  (λri. ∃r ys. size_rel ri r ★ arr_assn xsi ys ★
   ★ r ≤ n ★ length ys = length xs ★ drop r ys = drop r xs)
```

Where `htriple` is the Hoare triple for LLVM programs and `★` is the separating conjunction. This matches the specification of POSIX's `fread` function [30], except that we do not specify what data is read. This is sound, as it is a valid over-approximation of the actual behaviour.

4.7 Soundness Theorem

Finally, we generate an implementation of `read_check_lrat` (Sect. 4.5), obtaining:

```
read_check_lrat_impl :: 8 word ptr × 64 word ⇒ 1 word lLM
read_check_lrat_impl, read_check_lrat :: inp_assn → bool1_rel
```

Here, *inp_assn* implements the input string by an array and its length. In order to smoothly interface this function from C/C++, we eliminate the tuple type and return a byte instead of a bit. We define:

```

lrat_checker :: 8 word ptr ⇒ 64 word ⇒ 8 word LLVM
lrat_checker p n ≡
  if read_check_lrat_impl (p,n) then return 1 else return 0

```

Isabelle LLVM’s code generator creates LLVM code, and a matching header file:

```

export_llvm lrat_checker is uint8_t lrat_checker(uint8_t *, int64_t)
file ../code/lrat_checker_export.{ll,h}

```

We link this with a small C program that reads the command line, memory-maps the formula file, provides the function *fread_from_certificate* (cf. Sect. 4.6), calls the verified checker, and prints the result.

Chaining together the correctness of the functional checker (Theorem 2) and the refinement theorem for *read_check_lrat*, and unfolding some definitions yields:

Theorem 3 (Soundness of Implementation). *When we pass the checker a pointer cp to an array of size $cszi$ holding the bytes c , then the checker will terminate with the array being unchanged, and if the result is not zero, the bytes c in the array are a syntactically correct encoding of an unsatisfiable CNF:*

$$\begin{aligned}
 & \text{htriple } (arr_assn\ cp\ c\ \star\ size_rel\ cszi\ csz\ \star\ csz=length\ c) \\
 & (lrat_checker\ cp\ cszi) \\
 & (\lambda r. arr_assn\ cp\ c\ \star\ (r \neq 0 \implies (\exists F. (c,F) \in g_dimacs \wedge \neg sat\ F)))
 \end{aligned}$$

Note that this theorem does not depend on any complex data structures or refinements. Apart from the basic notions of Hoare triples, separation logic, machine words, and pointers to arrays, it only depends on our semantics of formulas (Sect. 2.1), and our grammar for the DIMACS format (Sect. 2.2).

5 Benchmarks

For our benchmarks, we have used the latest stable versions of the tools available at the time of writing: CaDiCaL 1.9.4 [7], *lrat-trim* 0.2.0 [27], *cake_lpr* 7a207e9 [8], *gratchk* dc6dd9d [15], *lrat-check* 9ee016c [12], and *lrat-acl2* (incremental) 8.5 [1] on *gcl* 2.6.13pre [13]. We used an AMD Ryzen 9 7950X3D machine with 128 GB DDR5 RAM and a 2.0 TB Samsung 990 Pro SSD disk.

We have used problems from the 2022 SAT competition³ [33]: out of the 156 problems proved unsatisfiable in the main track, CaDiCaL timed out on 5 after 5000s. The remaining 151 problems form our benchmark set.

³ We did not choose the 2023 competition, because the problems there are biased towards checkers that use techniques not available for direct LRAT generation in CaDiCaL.

First, we let the checker run in parallel to CaDiCaL, streaming the certificate directly into the checker. We used our checker and `cake_lpr`⁴. We measure the computing times (the sum of user and system

Checker	n	t_c/t_s	l_s	l_c	m_c/m_s	w/w_f	w/w_b
Our tool	151	6%	97%	5%	80%	102%	110%
<code>cake_lpr</code>	138	61%	85%	47%	162×	130%	143%

Table 1. Benchmark results in streaming mode. The table displays the averages over the successfully certified problems (n).

time) that were allocated to the sat solver (t_s) and checker (t_c). The ratio t_c/t_s indicates how the work is distributed between solver and checker. The smaller this ratio, the less time the checker needs in comparison to the solver. Next, we measure the average CPU loads allocated to the solver (l_s) and checker (l_c). A solver load less than 100% indicates that the solver was slowed down. The less load the checker produces, the fewer additional computing power is needed for checking. We also measure the peak memory consumption (maximum resident set size) of the solver (m_s) and checker (m_c). The ratio m_c/m_s indicates the additional memory required for checking. Finally, we measure the wall-clock time until certification finishes (w), and compare that to the time required by the solver to solve the problem and write the certificate to a file (w_f), and to the solving time without producing a certificate at all (w_b). The ratios w/w_f and w/w_b indicate the observed extra time required for certification. The results are displayed in Table 1: Our checker verified all problems, adding about 6% more computation time and 80% more memory on top of solving and certificate producing. It does not significantly slow down the solver, which runs at 97% CPU load. Compared to writing the certificate to a file, streaming it directly to the checker is 2% slower, and the overhead added by the whole certification process is 10%. The `cake_lpr` checker failed to certify 13 problems⁵. For the remaining problems, it added 61% of computation time, and the solver only ran at 85% load. Streaming the certificate to `cake_lpr` is 30% slower than writing the certificate to a file, and 43% slower than solving without producing a certificate. Moreover, for each `cake_lpr` run, maximum heap and stack sizes have to be determined upfront, and `cake_lpr` is likely to use all available heap⁶. Without prior knowledge of the problem, it is impossible to guess good sizes. For our experiments, we used 8 GiB stack and 16 GiB heap, based on the maximum of 11 GiB that our tool needed. With this, `cake_lpr` ran out of memory for six problems, and maxed out at around 16 GiB memory usage for most of the remaining problems (131/138). On average, it needed 162 times more memory than the solver.

⁴ We didn't include a Coq based `lrat`-checker [9], nor an ACL2 based one [16]: the former is reportedly less efficient than `cake_lpr` [35], and the latter supports, to the best of our knowledge, no streaming of the certificate.

⁵ 6 memouts, 6 parsing errors, most likely due to benchmarks incompatible with `CakeLPR`'s strict interpretation of the DIMACS CNF format, and one timeout at 5000 s.

⁶ We assume that the garbage collector only becomes active when available memory has filled up.

Checker	n	t_{tot}	t_{avg}	m_{avg}
lrat-trim	151	116%	118%	96%
lrat-check	150	357%	384%	116%
gratchk	147	917%	994%	80×
cake_lpr	138	1666%	1797%	208×
lrat-acl2	57	105×	8200%	158×

Table 2. Benchmark results in file mode.

For the garbage collected tools (gratchk, cake_lpr, lrat-acl2), we set a heap limit of 16GiB. If possible, we used binary LRAT encoding (our tool and lrat-trim), and did not include conversion time from LRAT to GRAT (gratchk). Using our tool as baseline (100%), we display the ratio of the total computation times over all problems (t_{tot}), and the average ratios of computation time and peak memory usage per problem (t_{avg} and m_{avg}). The results are displayed in Table 2: our tool is slightly faster but uses slightly more memory than lrat-trim. It is significantly faster and uses less memory than any other verified or unverified tool we tested. After 14:30h, lrat-acl2 had processed 66 problems and succeeded on 57. The same problems took 3:25m to check by our tool. We aborted the experiment at that point, as, by extrapolation, it would have taken 5 more days to complete.

6 Conclusion

We have used the Isabelle LLVM framework to formally verify soundness of an unsatisfiability certificate checker. Our checker is verified w.r.t. a grammar of the DIMACS format, a semantics of CNF, and down to the LLVM code that implements the checker. Completeness of the checker has been empirically verified by showing that it accepts a large set of benchmarks. Our checker accepts the LRUP fragment of the LRAT format, which makes it suitable for checking certificates from many top-performing SAT solvers. For solvers that support streaming of LRAT certificates, our tool can be run in parallel to the solver, eliminating the need to store the potentially large certificate, and coming back with the certification result the moment the solver is finished. For CaDiCaL, this is only 10% slower than running just the solver, and 2% slower than writing the certificate to a file without checking it. Our implementation is slightly faster and uses only 4% more memory than the unverified and highly optimized lrat-trim checker. It is significantly faster and more memory efficient than any other LRAT checker we know of, verified or unverified. This makes it possible to routinely run the checker with the solver, increasing the confidence at low cost.

To design our checker, we first implemented and profiled prototypes in C++ to determine the important optimizations. This took roughly 40 person hours. We then used the Isabelle Refinement Framework to produce a verified version of the checker. This was done in a top-down refinement process, which was guided by the experience from the unverified prototypes. This took another 200 h.

To measure the performance of just the checker, we ran it on certificates stored in files. For this experiment we also included the grattck tool, which is reported to be faster than cake_lpr [35], the lrat-acl2 tool, and the unverified checker implementations lrat-trim (forward) and lrat-check, to compare our verified tool against unverified but highly optimized implementations.

6.1 Related Work

The closest work to ours is the verified `cake_lpr` checker [34,35]. It supports streaming certificates⁷ and the full LPR format. The `cake_lpr` checker is verified down to assembly code (with a thin C wrapper around it), while our checker is verified down to LLVM intermediate code. While verifying an LLVM compiler is orthogonal to this project, we would immediately profit from such a verified compiler, further reducing our trusted code base. Moreover, our checker is verified w.r.t. a grammar of DIMACS CNF, while `cake-lpr`'s parser is not verified. It only comes with a sanity check, showing that the parser is left inverse to a pretty printer. Our checker is significantly faster than `cake_lpr`, and only allocates as much memory as needed, while `cake-lpr`'s memory size has to be set upfront, making it uncontrollable without background information about the problem. In particular in streaming mode, such information is not available. Finally, `cake_lpr` uses the ASCII encoding of LRAT, while our checker uses the more compact binary encoding.⁸

There are other verified certificate checkers [5,9,16,23], which, however, do not support streaming or are significantly slower than `cake_lpr`.

6.2 Future Work

There are no principle problems to extend our tool to the more powerful LRAT and LPR formats. We leave this to future work, as we are not aware of any solver that would support streaming these formats.

While our parser was manually implemented and then verified, there is work on verified parser generators [6,18–20,25,31]. We leave it to future work to integrate similar techniques into the Isabelle LLVM workflow.

While faster than parsing the ASCII encoding, decompression of the binary encoding is a hot-spot in our checker. In streaming mode, we could probably use a less compact but faster to read format, which we leave to future work.

References

1. ACL2 github repository. <https://github.com/acl2/acl2>
2. Baek, S., Carneiro, M., Heule, M.J.H.: A flexible proof format for SAT solver-elaborator communication. In: TACAS 2021. LNCS, vol. 12651, pp. 59–75. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_4
3. Beeren, J., et al.: Finite machine word library. Archive of Formal Proofs, June 2016. https://isa-afp.org/entries/Word_Lib.html. Formal proof development
4. Bogaerts, B., Gocht, S., McCreesh, C., Nordström, J.: Certified symmetry and dominance breaking for combinatorial optimisation. *J. Artif. Intell. Res.* **77**, 1539–1589 (2023). Preliminary version in AAAI 2022

⁷ Surprisingly, we have not found reports on using `cake_lpr` in streaming mode. In particular, Pollit et. al. [29] did not consider this possibility when they extended CaDiCaL to directly produce LRUP certificates.

⁸ Conversion between the encodings is easy, and we leave native support of the ASCII encoding in our checker to future work.

5. Bogaerts, B., McCreesh, C., Myreen, M.O., Nordström, J., Oertel, A., Tan, Y.K.: VeriPB and CakePB in the SAT competition 2023. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2023)
6. Bortin, M.: A formalisation of the Cocke-Younger-Kasami algorithm. Archive of Formal Proofs, April 2016. <https://isa-afp.org/entries/CYK.html>. Formal proof development
7. CaDiCaL github repository. <https://github.com/arminbiere/cadical/releases/tag/rel-1.9.4>
8. cake_lpr github repository. https://github.com/tanyongkiam/cake_lpr
9. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
10. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_7
11. Dawson, J.: Isabelle theories for machine words. Electron. Notes Theoret. Comput. Sci. **250**(1), 55–70 (2009). <https://doi.org/10.1016/j.entcs.2009.08.005>, <https://www.sciencedirect.com/science/article/pii/S1571066109003302>. Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007)
12. DRAT-trim github repository. <https://github.com/marijnheule/drat-trim>
13. GNU common lisp. <git://git.sv.gnu.org/gcl.git>
14. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, 2–4 January 2008 (2008). http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf
15. gratchk github repository. <https://github.com/IsaFoL/IsaFoL/tree/master/GRAT/gratchk>
16. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
17. Heule, M., Hunt, W., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design, FMCAD 2013, pp. 181–188. IEEE (2013)
18. Jia, X., Kumar, A., Tan, G.: A derivative-based parser generator for visibly push-down grammars. Proc. ACM Program. Lang. **5**(OOPSLA), 1–24 (2021). <https://doi.org/10.1145/3485528>
19. Jourdan, J.-H., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
20. Koprowski, A., Binsztok, H.: TRX: a formally verified parser interpreter. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 345–365. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_19

21. Lammich, P.: Refinement to Imperative/HOL. In: ITP, LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015)
22. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O’Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, 9–12 September 2019, Portland, OR, USA. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPIcs.ITP.2019.22>
23. Lammich, P.: Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.* **64**(3), 513–532 (2020). <https://doi.org/10.1007/s10817-019-09525-z>
24. Lammich, P., Tuerk, T.: Applying data refinement for Monadic programs to Hopcroft’s algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12
25. Lasser, S., Casinghino, C., Fisher, K., Roux, C.: Costar: a verified all(*) parser. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 420–434. PLDI 2021. Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3453483.3454053>
26. Lattner, C., Adev, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004, CGO 2004, pp. 75–86 (2004). <https://doi.org/10.1109/CGO.2004.1281665>
27. `lrat-trim` github repository. <https://github.com/arminbiere/lrat-trim/releases/tag/rel-0.2.0>
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of DAC, pp. 530–535. ACM (2001)
29. Pollitt, F., Fleury, M., Biere, A.: Faster LRAT checking than solving with CaDiCaL. In: Mahajan, M., Slivovsky, F. (eds.) 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, 4–8 July 2023, Alghero, Italy. LIPIcs, vol. 271, pp. 21:1–21:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPIcs.SAT.2023.21>
30. The Open Group Base Specifications (2018). Issue 7 (IEEE Std 1003.1-2017)
31. Rau, M.: Earley parser. Archive of Formal Proofs, July 2023. https://isa-afp.org/entries/Earley_Parser.html. Formal proof development
32. SAT competition 2009—submission format (2009). <http://www.satcompetition.org/2009/format-benchmarks2009.html>
33. SAT competition (2022). <https://satcompetition.github.io/2022/>
34. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: `cake_lpr`: verified propagation redundancy checking in CakeML. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 223–241. Springer, Cham (2021)
35. Tan, Y.K., Heule, M.J., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in CakeML. *Int. J. Softw. Tools Technol. Transfer* **25**(2), 167–184 (2023)
36. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31

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

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





Generalized Optimization Modulo Theories

Nestan Tsiskaridze¹(✉) , Clark Barrett¹ , and Cesare Tinelli² 

¹ Stanford University, Stanford, CA, USA

{nestan,barrett}@cs.stanford.edu

² The University of Iowa, Iowa City, IA, USA

cesare-tinelli@uiowa.edu

Abstract. Optimization Modulo Theories (OMT) has emerged as an important extension of the highly successful Satisfiability Modulo Theories (SMT) paradigm. The OMT problem requires solving an SMT problem with the restriction that the solution must be optimal with respect to a given objective function. We introduce a generalization of the OMT problem where, in particular, objective functions can range over partially ordered sets. We provide a formalization of and an abstract calculus for the Generalized OMT problem and prove their key correctness properties. Generalized OMT extends previous work on OMT in several ways. First, in contrast to many current OMT solvers, our calculus is theory-agnostic, enabling the optimization of queries over any theories or combinations thereof. Second, our formalization unifies both single- and multi-objective optimization problems, allowing us to study them both in a single framework and facilitating the use of objective functions that are not supported by existing OMT approaches. Finally, our calculus is sufficiently general to fully capture a wide variety of current OMT approaches (each of which can be realized as a specific strategy for rule application in the calculus) and to support the exploration of new search strategies. Much like the original abstract DPLL(T) calculus for SMT, our Generalized OMT calculus is designed to establish a theoretical foundation for understanding and research and to serve as a framework for studying variations of and extensions to existing OMT methodologies.

Keywords: Optimization Modulo Theories (OMT) · Optimization · Satisfiability Modulo Theories (SMT) · Abstract Calculus

1 Introduction

Over the past decade, the field of Optimization Modulo Theories (OMT) has emerged, inspiring the interest of researchers and practitioners alike. OMT builds on the highly successful Satisfiability Modulo Theories (SMT) [3] paradigm and extends it: while the latter focuses solely on finding a theory model for a first-order formula, the former adds an objective term that must be optimized with respect to some total ordering over the term's domain.

The development of OMT solvers has fostered research across an expanding spectrum of applications, including scheduling and planning with resources [7,

13, 17, 20, 26, 30, 35, 38, 48, 58], formal verification and model checking [37, 49], program analysis [10, 23, 25, 28, 69], requirements engineering and specification synthesis [21, 41–43], security analysis [4, 18, 46, 61], system design and configuration [14, 15, 29, 34, 47, 51, 63, 68], machine learning [59, 62], and quantum annealing [5].

Various OMT procedures have been developed for different types of optimization objectives (e.g., single- and multi-objective problems), underlying theories (e.g., arithmetic and bitvectors), and search strategies (e.g., linear and binary search). We provide an overview of established OMT techniques in Sect. 5. An extensive survey can be found in Trentin [64].

We introduce a proper generalization of the OMT problem and an abstract calculus for this generalization whose main goal is similar to that of the DPLL(T) calculus for SMT [45]: to provide both a foundation for theoretical understanding and research and a blueprint for practical implementations. Our approach is general in several ways. First, in contrast to previous work in OMT, it is parameterized by the optimization order, which does not need to be total, and it is not specific to any theory or optimization technique, making the calculus easily applicable to new theories or objective functions. Second, it encompasses both single- and multi-objective optimization problems, allowing us to study them in a single, unified framework and enabling combinations of objectives not covered in previous work. Third, it captures a wide variety of current OMT approaches, which can be realized as instances of the calculus together with specific strategies for rule application. Finally, it provides a framework for the exploration of new optimization strategies.

Contributions . To summarize, our contributions include:

- a formalization of a generalization of OMT to partial orders that unifies traditional single- and multi-objective optimization problems;
- a theory-agnostic abstract calculus for Generalized OMT that can also be used to describe and study previous OMT approaches;
- a framework for understanding and exploring search strategies for Generalized OMT; and
- proofs of correctness for important properties of the calculus.

The rest of the paper is organized as follows. Section 2 introduces background and notation. Section 3 defines the Generalized OMT problem. Section 4 presents the calculus, provides an illustrative example of its use and addresses its correctness¹. Finally, Sect. 5 discusses related work, and Sect. 6 concludes.

2 Background

We assume the standard many-sorted first-order logic setting for SMT, with the usual notions of signature, term, formula, and interpretation. We write $\mathcal{I} \models \phi$

¹ Full proofs and an additional example are provided in the longer version of this paper [67].

Table 1. Theory-specific notation.

Syntax	Semantics	Meaning
$\text{Bool}, \text{Int}, \text{Real}, BV_{[n]}, \text{Str}$		Sorts for Booleans, integers, reals, bitvectors of length n , and character strings
$+, -, \times, \div$		Arithmetic operators over reals/integers
$\langle_{\mathbf{R}}, \rangle_{\mathbf{R}}, \leq_{\mathbf{R}}, \geq_{\mathbf{R}}$	$\prec_{\mathbf{R}}, \succ_{\mathbf{R}}, \preceq_{\mathbf{R}}, \succeq_{\mathbf{R}}$	Comparison operators over reals
$\langle_{\text{Int}}, \rangle_{\text{Int}}, \leq_{\text{Int}}, \geq_{\text{Int}}$	$\prec_{\text{Int}}, \succ_{\text{Int}}, \preceq_{\text{Int}}, \succeq_{\text{Int}}$	Comparison operators over integers
$+_{[n]}, -_{[n]}, \times_{[n]}, \div_{[n]}$		Arithmetic modulo 2^n operators
$\langle_{[n]}, \rangle_{[n]}, \leq_{[n]}, \geq_{[n]}$	$\prec_{[n]}, \succ_{[n]}, \preceq_{[n]}, \succeq_{[n]}$	(Unsigned) comparison operators over $BV_{[n]}$ terms
$\text{ite}(c, x, y)$		If-then-else operator (if c then x else y)
$\text{tup}(t_1, \dots, t_n)$		n -ary tuple where element i is t_i
$\langle_{\text{str}}, \rangle_{\text{str}}, \leq_{\text{str}}, \geq_{\text{str}}$	$\prec_{\text{str}}, \succ_{\text{str}}, \preceq_{\text{str}}, \succeq_{\text{str}}$	Strict and non-strict lexicographic and reverse lexicographic orders on strings
ϵ		The empty string
$x \cdot y$		String concatenation operator
$\text{len}(x)$		String length operator
$\text{contains}(x, y)$		String containment operator (true iff y is a substring of x)

to mean that formula ϕ holds in or is *satisfied* by an interpretation \mathcal{I} . A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and \mathbf{I} is a class of Σ -interpretations. We call the elements of \mathbf{I} \mathcal{T} -interpretations. We write $\Gamma \models_{\mathcal{T}} \phi$, where Γ is a formula (or a set of formulas), to mean that Γ \mathcal{T} -entails ϕ , i.e., every \mathcal{T} -interpretation that satisfies (each formula in) Γ satisfies ϕ as well. For convenience, for the rest of the paper, we fix a background theory \mathcal{T} with equality and with signature Σ . We also fix an infinite set \mathcal{X} of sorted variables with sorts from Σ and assume $\prec_{\mathcal{X}}$ is some total order on \mathcal{X} . We assume that all terms and formulas are Σ -terms and Σ -formulas with free variables from \mathcal{X} . Since the theory \mathcal{T} is fixed, we will often abbreviate $\models_{\mathcal{T}}$ as \models and consider only interpretations that are \mathcal{T} -interpretations assigning a value to every variable in \mathcal{X} . At various places in the paper, we use sorts and operators from standard SMT-LIB theories such as integers, bitvectors, strings,² or data types [2]. We assume that every \mathcal{T} -interpretation interprets them in the same (standard) way. Table 1 lists theory symbols used in this paper and their meanings. A Σ -formula ϕ is *satisfiable* (resp., *unsatisfiable*) in \mathcal{T} if it is satisfied by some (resp., no) \mathcal{T} -interpretation.

Let s be a Σ -term. We denote by $s^{\mathcal{I}}$ the value of s in an interpretation \mathcal{I} , defined as usual by recursively determining the values of sub-terms. We denote by $FV(s)$ the set of all variables occurring in s . Similarly, we write $FV(\phi)$ to denote the set of all the free variables occurring in a formula ϕ . If $FV(\phi) = \{v_1, \dots, v_n\}$, where for each $i \in [1, n]$, $v_i \prec_{\mathcal{X}} v_{i+1}$, then the relation defined by ϕ (in \mathcal{T}) is $\{(v_1^{\mathcal{I}}, \dots, v_n^{\mathcal{I}}) \mid \mathcal{I} \models \phi \text{ for some } \mathcal{T}\text{-interpretation } \mathcal{I}\}$. A relation is *definable* in \mathcal{T} if there is some formula that defines it. Let \mathbf{v} be a tuple of variables (v_1, \dots, v_n) ,

² For simplicity, we assume strings are over characters ranging only from ‘a’ to ‘z’.

and let $\mathbf{t} = (t_1, \dots, t_n)$ be a tuple of Σ -terms, such that t_i and v_i are of the same sort for $i \in [1, n]$; then, we denote by $s[\mathbf{v} \leftarrow \mathbf{t}]$ the term obtained from s by simultaneously replacing each occurrence of variable v_i in s with the term t_i .

If S is a finite *sequence* (s_1, \dots, s_n) , we write $\text{TOP}(S)$ to denote, s_1 , the first element of S in S ; we write $\text{POP}(S)$ to denote the subsequence (s_2, \dots, s_n) of S . We use \emptyset to denote both the empty set and the empty sequence. We write $s \in S$ to mean that s occurs in the sequence S , and write $S \circ S'$ for the sequence obtained by appending S' at the end of S .

We adopt the standard notion of *strict partial order* \prec on a set A , that is, a relation in $A \times A$ that is irreflexive, asymmetric, and transitive. The relation \prec is a *strict total order* if, in addition, $a_1 \prec a_2$ or $a_2 \prec a_1$ for every pair a_1, a_2 of distinct elements of A . As usual, we will call \prec *well-founded* over a subset A' of A if A' contains no infinite descending chains. An element $m \in A$ is *minimal* (with respect to \prec) if there is no $a \in A$ such that $a \prec m$. If A has a unique minimal element, it is called a *minimum*.

3 Generalized Optimization Modulo Theories

We introduce a formalization of the *Generalized Optimization Modulo Theories* problem which unifies single- and multi-objective optimization problems and lays the groundwork for the calculus presented in Sect. 4.

3.1 Formalization

For the rest of the paper, we fix a theory \mathcal{T} with some signature Σ .

Definition 1 (Generalized Optimization Modulo Theories (GOMT)).
A Generalized Optimization Modulo Theories problem is a tuple $\mathcal{GO} := \langle t, \prec, \phi \rangle$, where:

- t , a Σ -term of some sort σ , is an objective term to optimize;
- \prec is a strict partial order definable in \mathcal{T} , whose defining formula has two free variables, each of sort σ ; and
- ϕ is a Σ -formula.

For any GOMT problem \mathcal{GO} and \mathcal{T} -interpretations \mathcal{I} and \mathcal{I}' , we say that:

- \mathcal{I} is \mathcal{GO} -consistent if $\mathcal{I} \models \phi$;
- \mathcal{I} \mathcal{GO} -dominates \mathcal{I}' , denoted by $\mathcal{I} <_{\mathcal{GO}} \mathcal{I}'$, if \mathcal{I} and \mathcal{I}' are \mathcal{GO} -consistent and $t^{\mathcal{I}} \prec t^{\mathcal{I}'}$; and
- \mathcal{I} is a \mathcal{GO} -solution if \mathcal{I} is \mathcal{GO} -consistent and no \mathcal{T} -interpretation \mathcal{GO} -dominates \mathcal{I} .

Informally, the term t represents the *objective function*, whose value we want to optimize. The order \prec is used to compare values of t , with a value a being considered *better* than a value a' if $a \prec a'$. Finally, the formula ϕ imposes constraints on the values that t can take. It is easy to see that the value of

$t^{\mathcal{I}}$ assigned by a \mathcal{GO} -solution \mathcal{I} is always minimal. As a special case, if \prec is a total order, then $t^{\mathcal{I}}$ is also unique (i.e., it is a minimum). Once we have fixed a GOMT problem \mathcal{GO} , we will informally refer to a \mathcal{GO} -consistent interpretation as a *solution* (of ϕ) and to a \mathcal{GO} -solution as an *optimal solution*.

Our notion of Generalized OMT is closely related to one by Bigarella et al. [6], which defines a notion of OMT for a generic background theory using a predicate that corresponds to a total order in that theory. Definition 1 generalizes this in two ways. First, we allow partial orders, with total orders being a special case. One useful application of this generalization is the ability to model multi-objective problems as single-objective problems over a suitable partial order, as we explain below. Second, we do not restrict \prec to correspond to a predicate symbol in the theory. Instead, any partial order *definable* in the theory can be used. This general framework captures a large class of optimization problems.

Example 1. Suppose \mathcal{T} is the theory of real arithmetic with the usual signature. Let $\mathcal{GO} := \langle x + y, \prec, 0 < x \wedge xy = 1 \rangle$, where x and y are variables of sort *Real* and \prec is defined by the formula $v_1 <_{\mathbb{R}} v_2$ (where $v_1 \prec_{\mathcal{X}} v_2$). A \mathcal{GO} -solution is any interpretation that interprets x and y as 1.

Example 2. With \mathcal{T} now being the theory of integer arithmetic, let $\mathcal{GO} = \langle x, \prec, x^2 < 20 \rangle$, where x is of sort *Int*, and \prec is defined by $v_1 >_{\text{Int}} v_2$ (where $v_1 \prec_{\mathcal{X}} v_2$). A \mathcal{GO} -solution must interpret x as the maximum integer satisfying $x^2 < 20$ (i.e., x must have value 4).

The examples above are both instances of what previous work refers to as single-objective optimization problems [64], with the first example being a minimization and the second a maximization problem. The next example illustrates a less conventional ordering.

Note that from now on, to keep the exposition simple, we define partial orders \prec appearing in \mathcal{GO} problems only *semantically*, i.e., formally, but without giving a specific defining formula. However, it is easy to check that all orders used in this paper are, in fact, definable in a suitable \mathcal{T} .

Example 3. Let $\mathcal{GO} = \langle x, \prec, x^2 < 20 \rangle$ be a variation of Example 2, where now, for any integers a and b , $a \prec b$ iff $|b| \prec_{\text{Int}} |a|$. A \mathcal{GO} -solution can interpret x either as 4 or -4 . Neither solution dominates the other since their absolute values are equal.

We next show how multi-objective problems are also instances of Definition 1.

3.2 Multi-objective Optimization

We use the term *multi-objective optimization* to refer to an optimization problem consisting of several sub-problems, each of which is also an optimization problem. A multi-objective optimization may also require specific interrelations among its sub-problems. In this section, we define several varieties of multi-objective optimization problems and show how each can be realized using Definition 1.

For each, we also state a correctness proposition which follows straightforwardly from the definitions.

In the following, given a strict ordering \prec , we will denote its reflexive closure by \preceq . We start with a multi-objective optimization problem which requires that the sub-problems be prioritized in lexicographical order [8, 9, 53, 56, 64].

Definition 2 (Lexicographic Optimization (LO)). *A lexicographic optimization problem is a sequence of GOMT problems $\mathcal{LO} = (\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, where $\mathcal{GO}_i := \langle t_i, \prec_i, \phi_i \rangle$ for $i \in [1, n]$. For \mathcal{I} -interpretations \mathcal{I} and \mathcal{I}' , we say that:*

- \mathcal{I} \mathcal{LO} -dominates \mathcal{I}' , denoted by $\mathcal{I} <_{\mathcal{LO}} \mathcal{I}'$, if \mathcal{I} and \mathcal{I}' are \mathcal{GO}_i -consistent for each $i \in [1, n]$, and for some $j \in [1, n]$:
 - (i) $t_i^{\mathcal{I}} = t_i^{\mathcal{I}'}$ for all $i \in [1, j]$; and
 - (ii) $t_j^{\mathcal{I}} \prec_j t_j^{\mathcal{I}'}$.
- \mathcal{I} is a solution to \mathcal{LO} iff \mathcal{I} is \mathcal{GO}_i -consistent for each i and no \mathcal{I} -interpretation \mathcal{LO} -dominates \mathcal{I} .

An \mathcal{LO} problem can be solved by converting it into an instance of Definition 1.

Definition 3 ($\mathcal{GO}_{\mathcal{LO}}$). *Given an \mathcal{LO} problem $(\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, with $\mathcal{GO}_i := \langle t_i, \prec_i, \phi_i \rangle$ for $i \in [1, n]$, the corresponding \mathcal{GO} instance is defined as $\mathcal{GO}_{\mathcal{LO}}(\mathcal{GO}_1, \dots, \mathcal{GO}_n) := \langle t, \prec_{\mathcal{LO}}, \phi \rangle$, where:*

- $t = \text{tup}(t_1, \dots, t_n)$; $\phi = \phi_1 \wedge \dots \wedge \phi_n$;
- if t is of sort σ , then $\prec_{\mathcal{LO}}$ is the lexicographic extension of $(\prec_1, \dots, \prec_n)$ to σ^T : for $(a_1, \dots, a_n), (b_1, \dots, b_n) \in \sigma^T$, $(a_1, \dots, a_n) \prec_{\mathcal{LO}} (b_1, \dots, b_n)$ iff for some $j \in [1, n]$:
 - (i) $a_i = b_i$ for all $i \in [1, j]$; and
 - (ii) $a_j \prec_j b_j$.

Here and in other definitions below, we use the data type theory constructor tup to construct the objective term t . This is a convenient mechanism for keeping an ordered list of the sub-objectives and keeps the overall theoretical framework simple. In practice, if using a solver that does not support tuples or the theory of data types, other implementation mechanisms could be used. Note that if each sub-problem uses a total order, then $\prec_{\mathcal{LO}}$ will also be total.

Proposition 1. *Let \mathcal{I} be a $\mathcal{GO}_{\mathcal{LO}}$ -solution. Then \mathcal{I} is also a solution to the corresponding \mathcal{LO} problem as defined in Definition 2.*

Example 4 (\mathcal{LO}). Let $\mathcal{GO}_1 := \langle x, \prec_1, \text{True} \rangle$ and $\mathcal{GO}_2 := \langle y +_{[2]} z, \prec_2, \text{True} \rangle$, where x, y, z are variables of sort $BV_{[2]}$, $a \prec_1 b$ iff $a \prec_{[2]} b$, and $a \prec_2 b$ iff $a \succ_{[2]} b$. Now, let $\mathcal{GO} = \mathcal{GO}_{\mathcal{LO}}(\mathcal{GO}_1, \mathcal{GO}_2) = \langle t, \prec_{\mathcal{LO}}, \text{True} \rangle$. Then, $t = \text{tup}(x, y +_{[2]} z)$ and $(a_1, a_2) \prec_{\mathcal{LO}} (b_1, b_2)$ iff $a_1 \prec_{[2]} b_1$ or $(a_1 = b_1$ and $a_2 \succ_{[2]} b_2)$. Now, let $\mathcal{I}, \mathcal{I}'$, and \mathcal{I}'' be such that: $x^{\mathcal{I}} = 11, y^{\mathcal{I}} = 00, z^{\mathcal{I}} = 10$, and $t^{\mathcal{I}} := (11, 10)$; $x^{\mathcal{I}'} = 01, y^{\mathcal{I}'} = 01, z^{\mathcal{I}'} = 01$, and $t^{\mathcal{I}'} := (01, 10)$; $x^{\mathcal{I}''} = 01, y^{\mathcal{I}''} = 01, z^{\mathcal{I}''} = 10$, and $t^{\mathcal{I}''} := (01, 11)$. Then, $\mathcal{I}'' <_{\mathcal{GO}} \mathcal{I}' <_{\mathcal{GO}} \mathcal{I}$, since $(01, 11) \prec_{\mathcal{LO}} (01, 10) \prec_{\mathcal{LO}} (11, 10)$.

We can also accommodate Pareto optimization [8,9,64] in our framework.

Definition 4 (Pareto Optimization (PO)). A Pareto optimization problem is a sequence of GOMT problems $\mathcal{PO} = (\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, where $\mathcal{GO}_i := \langle t_i, \prec_i, \phi_i \rangle$ for $i \in [1, n]$. For any \mathcal{T} -interpretations \mathcal{I} and \mathcal{I}' , we say that:

- \mathcal{I} \mathcal{PO} -dominates, or Pareto dominates, \mathcal{I}' , denoted by $\mathcal{I} <_{\mathcal{PO}} \mathcal{I}'$, if \mathcal{I} and \mathcal{I}' are \mathcal{GO} -consistent w.r.t. each \mathcal{GO}_i , $i \in [1, n]$, and:
 - (i) $t_i^{\mathcal{I}} \preceq_i t_i^{\mathcal{I}'}$ for all $i \in [1, n]$; and
 - (ii) for some $j \in [1, n]$, $t_j^{\mathcal{I}} \prec_j t_j^{\mathcal{I}'}$.
- \mathcal{I} is a solution to \mathcal{PO} iff \mathcal{I} is \mathcal{GO} -consistent w.r.t. each \mathcal{GO}_i and no \mathcal{I}' \mathcal{PO} -dominates \mathcal{I} .

Definition 5 ($\mathcal{GO}_{\mathcal{PO}}$). Given a PO problem $\mathcal{PO} = (\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, we define $\mathcal{GO}_{\mathcal{PO}}(\mathcal{GO}_1, \dots, \mathcal{GO}_n) := \langle t, \prec_{\mathcal{PO}}, \phi \rangle$, where:

- $t = \text{tup}(t_1, \dots, t_n)$; $\phi = \phi_1 \wedge \dots \wedge \phi_n$;
- if t is of sort σ , then $\prec_{\mathcal{PO}}$ is the pointwise extension of $(\prec_1, \dots, \prec_n)$ to $\sigma^{\mathcal{T}}$; for any $(a_1, \dots, a_n), (b_1, \dots, b_n) \in \sigma^{\mathcal{T}}$, $(a_1, \dots, a_n) \prec_{\mathcal{PO}} (b_1, \dots, b_n)$ iff:
 - (i) $a_i \preceq_i b_i$ for all $i \in [1, n]$; and
 - (ii) $a_j \prec_j b_j$ for some $j \in [1, n]$.

Proposition 2. Let \mathcal{I} be a $\mathcal{GO}_{\mathcal{PO}}$ -solution. Then \mathcal{I} is also a solution to the corresponding \mathcal{PO} problem as defined in Definition 4.

Next, consider a \mathcal{PO} example with two sub-problems: one minimizing the length of a string w , and the other maximizing a substring x of w lexicographically.

Example 5 (\mathcal{PO}). Let \mathcal{T} be the SMT-LIB theory of strings and let $\mathcal{GO}_1 := \langle \text{len}(w), \prec_1, \text{len}(w) < 4 \rangle$ and $\mathcal{GO}_2 := \langle x, \prec_2, \text{contains}(w, x) \rangle$, where w, x are variables of sort Str , \prec_1 is \prec_{Int} , and \prec_2 is \succ_{Str} . Now, let $\mathcal{GO}_{\mathcal{PO}} = \mathcal{GO}_{\mathcal{PO}}(\mathcal{GO}_1, \mathcal{GO}_2) = \langle t, \prec_{\mathcal{PO}}, \text{len}(w) < 4 \wedge \text{contains}(x, w) \rangle$. Then, $t = \text{tup}(\text{len}(w), x)$ and $(a_1, a_2) \prec_{\mathcal{PO}} (b_1, b_2)$ iff $a_1 \preceq_{\text{Int}} b_1$, $a_2 \succ_{\text{Str}} b_2$, and $(a_1 \prec_{\text{Int}} b_1$ or $a_2 \succ_{\text{Str}} b_2)$. Now, let $\mathcal{I}, \mathcal{I}'$, and \mathcal{I}'' be such that: $\mathcal{I} := \{w \mapsto \text{"aba"}, x \mapsto \text{"ab"}\}$ and $t^{\mathcal{I}} := (3, \text{"ab"})$; $\mathcal{I}' := \{w \mapsto \text{"z"}, x \mapsto \text{"z"}\}$ and $t^{\mathcal{I}'}$:= (1, "z"); and $\mathcal{I}'' := \{w \mapsto \epsilon, x \mapsto \epsilon\}$ and $t^{\mathcal{I}''}$:= (0, ϵ). Then, $\mathcal{I}' <_{\mathcal{GO}} \mathcal{I}$, since $(1, \text{"z"}) \prec_{\mathcal{PO}} (3, \text{"ab"})$; but both \mathcal{I} and \mathcal{I}' are incomparable with \mathcal{I}'' . Both \mathcal{I}' and \mathcal{I}'' are optimal solutions.

Though we omit them for space reasons, we can similarly capture the MinMax and MaxMin optimization problems [56,64] as corresponding $\mathcal{GO}_{\text{MINMAX}}$ and $\mathcal{GO}_{\text{MAXMIN}}$ instances of Definition 1.³

Note that except for degenerate cases, the orders used for MinMax and MaxMin, as well as the order $\prec_{\mathcal{PO}}$ above, are always partial orders. Being able to model these multi-objective optimization problems in a clean and simple way is a main motivation for using a partial instead of a total order in Definition 1.

Another problem in the literature is the *multiple-independent* (or *boxed*) optimization problem [8,9,64]. It simultaneously solves several independent GOMT problems. We show how to realize this as a single \mathcal{GO} instance.

³ Details of these formulations can be found in the longer version of this paper [67].

Definition 6 (Boxed Optimization (BO)). A boxed optimization problem is a sequence of GOMT problems, $\mathcal{BO} = (\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, where $\mathcal{GO}_i := \langle t_i, \prec_i, \phi_i \rangle$ for $i \in [1, n]$. We say that:

- A sequence of interpretations $(\mathcal{I}_1, \dots, \mathcal{I}_n)$ \mathcal{BO} -dominates $(\mathcal{I}'_1, \dots, \mathcal{I}'_n)$, denoted by $(\mathcal{I}_1, \dots, \mathcal{I}_n) <_{\mathcal{BO}} (\mathcal{I}'_1, \dots, \mathcal{I}'_n)$, if \mathcal{I}_i and \mathcal{I}'_i are \mathcal{GO}_i -consistent or each $i \in [1, n]$, and:
 - (i) $t_i^{\mathcal{I}_i} \prec_i t_i^{\mathcal{I}'_i}$ for all $i \in [1, n]$; and
 - (ii) for some $j \in [1, n]$, $t_j^{\mathcal{I}_j} \prec_j t_j^{\mathcal{I}'_j}$.
- $(\mathcal{I}_1, \dots, \mathcal{I}_n)$ is a solution to \mathcal{BO} iff \mathcal{I}_i is \mathcal{GO}_i -consistent for each $i \in [1, n]$ and no $(\mathcal{I}'_1, \dots, \mathcal{I}'_n)$ \mathcal{BO} -dominates $(\mathcal{I}_1, \dots, \mathcal{I}_n)$.

Note that in previous work, there is an additional assumption that $\phi_i = \phi_j$ for all $i, j \in [1, n]$. Below, we show how to solve the more general case without this assumption. We first observe that the above definition closely resembles Definition 4 for Pareto optimization (PO) problems. Leveraging this similarity, we show how to transform an instance of a BO problem into a PO problem.

Definition 7. ($\mathcal{GO}_{\mathcal{BO}}$) Let $\mathcal{BO} = (\mathcal{GO}_1, \dots, \mathcal{GO}_n)$, where $\mathcal{GO}_i := \langle t_i, \prec_i, \phi_i \rangle$ for $i \in [1, n]$. Let V_i be the set of all free variables in the i^{th} sub-problem that also appear in at least one other sub-problem:

$$V_i = (FV(t_i) \cup FV(\phi_i)) \cap \bigcup_{j \in [1, n], j \neq i} FV(t_j) \cup FV(\phi_j).$$

Let $\mathbf{v}_i = (v_{i,1}, \dots, v_{i,m})$ be some ordering of the variables in V_i (say, by \prec_X), and for each $j \in [1, m]$, let $v'_{i,j}$ be a fresh variable of the same sort as $v_{i,j}$, and let $\mathbf{v}'_i = (v'_{i,1}, \dots, v'_{i,m})$. Then, let $t'_i = t_i[\mathbf{v}_i \leftarrow \mathbf{v}'_i]$, $\phi'_i = \phi_i[\mathbf{v}_i \leftarrow \mathbf{v}'_i]$, and $\mathcal{GO}'_i = \langle t'_i, \prec_i, \phi'_i \rangle$. Then we define $\mathcal{GO}_{\mathcal{BO}} := \mathcal{GO}_{\mathcal{PO}}(\mathcal{GO}'_1, \dots, \mathcal{GO}'_n)$.

Proposition 3. Let \mathcal{I} be a solution to $\mathcal{GO}_{\mathcal{BO}}$ as defined in Definition 7. Then $(\mathcal{I}_1, \dots, \mathcal{I}_n)$ is a solution to the corresponding \mathcal{BO} problem as defined in Definition 6, where for each $i \in [1, n]$, \mathcal{I}_i is the same as \mathcal{I} except that each variable $v_{i,j} \in V_i$ is interpreted as $(v'_{i,j})^{\mathcal{I}}$.

In practice, solvers for BO problems can be implemented without variable renaming (see, e.g., [8, 36, 53]). Variable renaming, while a useful theoretical construct, also adds generality to our definition of \mathcal{BO} . An interesting direction for future experimental work would be to compare the two approaches in practice.

Compositional Optimization. GOMT problems can also be combined by functional composition of multiple objective terms, possibly of different sorts, yielding *compositional optimization problems* [12, 62, 64]. Our framework handles them naturally by simply constructing an objective term capturing the desired compositional relationship. For example, compositional objectives can address the (partial) MaxSMT problem [64], where some formulas are *hard* constraints and others are *soft* constraints. The goal is to satisfy all hard constraints and as many soft constraints as possible. The next example is inspired by Cimatti et al. [12] and Teso et al. [62].

Example 6 (MAXSMT). Let $x \geq 0$ and $y \geq 0$ be hard constraints and $4x + y - 4 \geq 0$ and $2x + 3y - 6 \geq 0$ soft constraints. We can formalize this as $\mathcal{GO}_{CO} = \langle t, \prec, \phi \rangle$, where: $t = ite(4x + y - 4 \geq 0, 0, 1) + ite(2x + 3y - 6 \geq 0, 0, 1)$, $\prec \equiv \prec_{Int}$, and $\phi = x \geq 0 \wedge y \geq 0$. An optimal solution must satisfy both hard constraints and, by minimizing the objective term t , as many soft constraints as possible.

MAXSMT has various variants including generalized, partial, weighted, and partial weighted MAXSMT [64], all of which our framework can handle similarly.

Next, we show a different compositional example that combines two different orders, one on strings and the other on integers. This example also illustrates a theory combination not present in the OMT literature.

Example 7 (Composition of Str and Int). Let \mathcal{T} be again the theory of strings⁴ Let $\mathcal{GO}_{CO} = \langle tup(x, len(x)), \prec, contains(x, "a") \wedge len(x) > 1 \rangle$, where x is of sort \mathbf{Str} and $(a_1, b_1) \prec (a_2, b_2)$ iff $b_1 \prec_{Int} b_2$ or $(b_1 = b_2$ and $a_1 \succ_{str} a_2)$. \prec prioritizes minimizing the length, but then maximizes the string with respect to lexicographic order. An optimal solution must interpret x as the string "za" of length 2 since x must be of length at least 2 and contain "a", making "za" the largest string of minimum length.

Based on the definitions given in this section, we see that our formalism can capture any combination of \mathcal{GO} (including compositional), \mathcal{GO}_{CO} , \mathcal{GO}_{PO} , \mathcal{GO}_{MINMAX} , \mathcal{GO}_{MAXMIN} , and \mathcal{GO}_{BO} problems. And note that the last four all make use of the partial order feature of Definition 1.

4 The GOMT Calculus

We introduce a calculus for solving the GOMT problem, presented as a set of *derivation rules*. We fix a GOMT problem $\mathcal{GO} = \langle t, \prec, \phi \rangle$ where ϕ is satisfiable (optimizing does not make sense otherwise). We start with a few definitions.

Definition 8 (State). *A state is a tuple $\Psi = \langle \mathcal{I}, \Delta, \tau \rangle$, where \mathcal{I} is an interpretation, Δ is a formula, and τ is a sequence of formulas.*

The set of all states forms the state space for the GOMT problem. Intuitively, the proof procedure of the calculus is a search procedure over this state space which maintains at all times a *current state* $\langle \mathcal{I}, \Delta, \tau \rangle$ storing a candidate solution and additional search information. In the current state, \mathcal{I} is the best solution found so far in the search; Δ is a formula describing the remaining, yet unexplored, part of the state space, where a better solution might exist; and τ contains formulas that divide up the search space described by Δ into *branches* represented by the individual formulas in τ , maintaining the invariant that the disjunction of all the formulas τ_1, \dots, τ_p in τ is equivalent to Δ modulo ϕ , that is, $\phi \models (\bigvee_{i=1}^p \tau_i \Leftrightarrow \Delta)$.

Note that states contain \mathcal{T} -interpretations, which are possibly infinite mathematical structures. This is useful to keep the calculus simple. In practice, it

⁴ The SMT-LIB theory of strings includes the theory of integers to support constraints over string length.

is enough just to keep track of the interpretations of the (finitely-many) symbols without fixed meanings (variables and uninterpreted functions and sorts) appearing in the state, much as SMT solvers do in order to produce models.

Definition 9 (Solve). *SOLVE is a function that takes a formula and returns a satisfying interpretation if the formula is satisfiable and a distinguished value \perp otherwise.*

Definition 10 (Better). *BETTER $_{\mathcal{GO}}$ is a function that takes a \mathcal{GO} -consistent interpretation \mathcal{I} and returns a formula BETTER $_{\mathcal{GO}}(\mathcal{I})$ with the property that for every \mathcal{GO} -consistent interpretation \mathcal{I}' ,*

$$\mathcal{I}' \models \text{BETTER}_{\mathcal{GO}}(\mathcal{I}) \text{ iff } \mathcal{I}' <_{\mathcal{GO}} \mathcal{I}.$$

The function above is specific to the given optimization problem \mathcal{GO} or, put differently, is parametrized by t , \prec , and ϕ . When \mathcal{GO} is clear, however, we simply write BETTER, for conciseness.

The calculus relies on the existence and computability of SOLVE and BETTER. SOLVE can be realized by any standard SMT solver. BETTER relies on a defining formula for \prec as discussed below. We note that intuitively, BETTER(\mathcal{I}) is simply a (possibly unsatisfiable) formula characterizing the solutions of ϕ that are better than \mathcal{I} . Assuming α_{\prec} is the formula defining \prec , with free variables $v_1 \prec_{\mathcal{X}} v_2$, if the value $t^{\mathcal{I}}$ can be represented by some constant c (e.g., if $t^{\mathcal{I}}$ is a rational number), then BETTER(\mathcal{I}) = $\alpha_{\prec}[(v_1, v_2) \leftarrow (t, c)]$ satisfies Definition 10. On the other hand, it could be that $t^{\mathcal{I}}$ is not representable as a constant (e.g., it could be an algebraic real number); then, a more sophisticated formula (involving, say, a polynomial and an interval specifying a particular root) may be required.

Definition 11 (Initial State). *The initial state of the GOMT problem $\mathcal{GO} = \langle t, \prec, \phi \rangle$ is $\langle \mathcal{I}_0, \Delta_0, \tau_0 \rangle$, where $\mathcal{I}_0 = \text{SOLVE}(\phi)$, $\Delta_0 = \text{BETTER}(\mathcal{I}_0)$, $\tau_0 = (\Delta_0)$.*

Note that $\mathcal{I}_0 \neq \perp$ since we assume that ϕ is satisfiable. The search for an optimal solution to the GOMT problem in our calculus starts with an arbitrary solution of the constraint ϕ and continues until it finds an optimal one.

4.1 Derivation Rules

Figure 1 presents the derivation rules of the GOMT calculus. The rules are given in guarded assignment form, where the rule premises describe the conditions on the current state that must hold for the rule to apply, and the conclusion describes the resulting modifications to the state. State components not mentioned in the conclusion of a rule are unchanged.

A derivation rule *applies* to a state if (i) the conditions in the premise are satisfied by the state and (ii) the resulting state is different. A state is *saturated* if no rules apply to it. A \mathcal{GO} -*derivation* is a sequence of states, possibly infinite, where the first state is the initial state of the GOMT problem \mathcal{GO} , and each state

$$\begin{array}{c}
\tau \neq \emptyset \quad \psi = \text{TOP}(\tau) \quad \phi \models \psi \Leftrightarrow \bigvee_{j=1}^k \psi_j, k \geq 1 \\
\text{F-SPLIT} \frac{}{\tau := (\psi_1, \dots, \psi_k) \circ \text{POP}(\tau)} \\
\text{F-SAT} \frac{\tau \neq \emptyset \quad \psi = \text{TOP}(\tau) \quad \text{SOLVE}(\phi \wedge \psi) = \mathcal{I}' \quad \mathcal{I}' \neq \perp \quad \Delta' = \Delta \wedge \text{BETTER}(\mathcal{I}')}{\mathcal{I} := \mathcal{I}', \Delta := \Delta', \tau := (\Delta')} \\
\text{F-CLOSE} \frac{\tau \neq \emptyset \quad \psi = \text{TOP}(\tau) \quad \text{SOLVE}(\phi \wedge \psi) = \perp}{\Delta := \Delta \wedge \neg\psi, \tau := \text{POP}(\tau)}
\end{array}$$

Fig. 1. The derivation rules of the GOMT Calculus.

in the sequence is obtained by applying one of the rules to the previous state. The *solution sequence* of a derivation is the sequence made up of the solutions (i.e., the interpretations) in each state of the derivation.

The calculus starts with a solution for ϕ and improves on it until an optimal solution is found. During a derivation, the best solution found so far is maintained in the \mathcal{I} component of the current state. A search for a better solution can be organized into branches through the use of the F-SPLIT rule. Progress toward a better solution is enforced by the formula Δ which, by construction, is falsified by all the solutions found so far. We elaborate on the individual rules next.

F-SPLIT. F-SPLIT divides the branch of the search space represented by the top formula $\psi = \text{TOP}(\tau)$ in τ into k sub-branches (ψ_1, \dots, ψ_k) , ensuring their disjunction is equivalent to ψ modulo the constraint ϕ : $\phi \models \psi \Leftrightarrow \bigvee_{j=1}^k \psi_j$. The rest of the state remains unchanged. F-SPLIT is applicable whenever τ is non-empty. The rule does not specify how the formulas ψ_1, \dots, ψ_k are chosen. However, a pragmatic implementation should aim to generate them so that they are *irredundant* in the sense that no formula is entailed modulo ϕ by the (disjunction of the) other formulas. This way, each branch potentially contains a solution that the others do not. Note, however, that this is not a requirement.

F-SAT. The F-SAT rule applies when there is a solution in the branch represented by the top formula ψ in τ . The rule selects a solution $\mathcal{I}' = \text{SOLVE}(\phi \wedge \psi)$ from that branch. One can prove that, by the way the formulas in τ are generated in the calculus, \mathcal{I}' necessarily improves on the current solution \mathcal{I} , moving the search closer to an optimal solution.⁵ Thus, F-SAT switches to the new solution (with $\mathcal{I} := \mathcal{I}'$) and directs the search to seek an even better solution by updating Δ to $\Delta' = \Delta \wedge \text{BETTER}(\mathcal{I}')$. Note that F-SAT resets τ to the singleton sequence (Δ') , discarding any formulas in τ . This is justified, as any discarded better solutions must also be in the space defined by Δ' .

F-CLOSE. The F-CLOSE rule eliminates the first element ψ of a non-empty τ if the corresponding branch contains no solutions (i.e., $\text{SOLVE}(\phi \wedge \psi) = \perp$).

⁵ See Lemma 6 in Appendix B of a longer version of this paper [67].

The rule further updates the state by adding the negation of ψ to Δ as a way to eliminate from further consideration the interpretations satisfying ψ .

Note that rules F-SAT and F-CLOSE both update Δ to reflect the remaining search space, whereas F-SPLIT refines the division of the current search space.

4.2 Search Strategies

The GOMT calculus provides the flexibility to support different search strategies. Here, we give some examples, including both notable strategies from the OMT literature as well as new strategies enabled by the calculus, and explain how they work at a conceptual level.

Divergence of Strategies: The strategies discussed below, with the exception of Hybrid search, may diverge if an optimal solution does not exist or if there is a *Zeno-style* [54, 55] infinite chain of increasingly better solutions, all dominated by an optimal one. We discuss these issues and termination in general in Sect. 4.4.

Linear Search: A linear search strategy is obtained by never using the F-SPLIT rule. Instead, the F-SAT rule is applied to completion (that is, repeatedly until it no longer applies). As we show later (see Theorem 2), in the absence of Zeno chains, τ eventually becomes empty, terminating the search. At that point, \mathcal{I} is guaranteed to be an optimal solution.

Binary Search: A binary search strategy is achieved by using the F-SPLIT rule to split the search space represented by $\psi = \text{TOP}(\tau)$ into two subspaces, represented by two formulas ψ_1 and ψ_2 , with $\phi \models \psi \Leftrightarrow (\psi_1 \vee \psi_2)$. In a strict binary search strategy, ψ_1 and ψ_2 should be chosen so that the two subspaces are disjoint and, to the extent possible, of equal size. A typical binary strategy alternates applications of F-SPLIT with applications of either F-SAT or F-CLOSE until τ becomes empty, at which point \mathcal{I} is guaranteed to be an optimal solution. A smart strategy would aim to find an optimal solution as soon as possible by arranging for solutions in ψ_1 (which will be checked first) to be better than solutions in ψ_2 , if this is easy to determine. Note that an unfortunate choice of ψ_1 by F-SPLIT, containing no solutions at all, is quickly remedied by an application of F-CLOSE which removes ψ_1 , allowing ψ_2 to be considered next. The same problem of Zeno-style infinite chains can occur in this strategy.

Multi-directional Exploration: For multi-objective optimization problems, a search strategy can be defined to simultaneously direct the search space towards any or all objectives. Formally, if n is the number of objectives, then the F-SPLIT rule can be instantiated in such a way that $\psi_j = \bigwedge_{i=1}^n \psi_{ji}$, where ψ_{ji} is a formula describing a part of the search space for the i^{th} objective term in the j^{th} branch.

Search Order: We formalize τ as a sequence to enforce exploring the branches in τ in a specific order, assuming such an order can be determined at the time of applying F-SPLIT. Often, this is the case. For example, in binary search, it is typically best to explore the section of the search space with better objective values first. If a solution is found in this section, a larger portion of the search space is pruned. Conversely, if the branches are explored in another order, even finding a solution necessitates continued exploration of the space corresponding to the remaining branches.

Alternatively, τ can be implemented as a set, by redefining the TOP and POP functions accordingly to select and remove a desired element in τ . With τ defined as a set, additional search strategies are possible, including parallel exploration of the search space and the ability to arbitrarily switch between branches.

Hybrid Search: For some objectives and orders, there exist off-the-shelf external optimization procedures (e.g., Simplex for linear real arithmetic). One way to integrate such a procedure into our calculus is to replace a call to the SOLVE function in F-SAT with a call to an external optimization procedure OPTIMIZE that is sort- and order-compatible with the GOMT problem. We pass to OPTIMIZE as parameters the constraint $\phi \wedge \text{TOP}(\tau)$ and the objective t and obtain an optimal solution in the current branch $\text{TOP}(\tau)$.⁶ The call can be viewed as an accelerator for a linear search on the current branch. This approach incorporates theory-specific optimization solvers in much the same way as is done in the OMT literature. However, our calculus extends previous approaches with the ability to blend theory-specific optimization with theory-agnostic optimization by interleaving applications of F-SAT using SOLVE with applications using OPTIMIZE. For example, we may want to alternate between expensive calls to an external optimization solver and calls to a standard solver that are guided by a custom branching heuristic.

Other Strategies: The calculus enables us to mix and match the above strategies arbitrarily, as well as to model other notable search techniques like cutting planes [16] by integrating a cut formula into SOLVE. And, of course, one advantage of an abstract calculus is that its generality provides a framework for the exploration of new strategies. Such an exploration is a promising direction for future work.

4.3 New Applications

A key feature of our framework is that it is theory-agnostic, that is, it can be used with any SMT theory or combination of theories. This is in contrast to most of the OMT literature in which a specific theory is targeted. It also fully supports arbitrary composition of GOMT problems using the multi-objective approaches described in Sect. 3.2. Thus, our framework enables OMT to be extended to new

⁶ This assumes there exists an optimal solution in the current branch. If not (i.e., if the problem is unbounded), a suitable error can be raised and the search terminated.

application areas requiring either combinations of theories or multi-objective formulations that are unsupported by previous approaches. We illustrate this (and the calculus itself) using a Pareto optimization problem over the theories of strings and integers (a combination of theories and objectives unsupported by any existing OMT approach or solver).

Example 8 ($\mathcal{GO}_{\mathcal{PO}}$). Let $\mathcal{GO}_1 := \langle \text{len}(w), \prec_1, \text{len}(s) < \text{len}(w) \rangle$ and $\mathcal{GO}_2 := \langle x, \prec_2, x = s \cdot w \cdot s \rangle$, where w, x, s are of sort **Str**, $\text{len}(w)$ and $\text{len}(s)$ are of sort **Int**, $\prec_1 \equiv \prec_{\text{Int}}$, and $\prec_2 \equiv \succ_{\text{Str}}$. Then, let $\mathcal{GO}_{\mathcal{PO}}(\mathcal{GO}_1, \mathcal{GO}_2) := \langle t, \prec_{\mathcal{PO}}, \phi \rangle$, where t is $\text{tup}(\text{len}(w), x)$, ϕ is $x = s \cdot w \cdot s \wedge \text{len}(s) < \text{len}(w)$, and $(a_1, a_2) \prec_{\mathcal{PO}} (b_1, b_2)$ iff $a_1 \preceq_1 b_1$, $a_2 \preceq_2 b_2$, and either $a_1 \prec_1 b_1$ or $a_2 \prec_2 b_2$ or both. Suppose initially:

$$\begin{aligned} \mathcal{I}_0 &= \{x \mapsto \text{"aabaa"}, s \mapsto \text{"a"}, w \mapsto \text{"aba"}, \}, & \tau_0 &= (\Delta_0), \\ \Delta_0 &= (\text{len}(w) \leq 3 \wedge x >_{\text{str}} \text{"aabaa"}) \vee (\text{len}(w) < 3 \wedge x \geq_{\text{str}} \text{"aabaa"}). \end{aligned}$$

The initial objective term value is $(3, \text{"aabaa"})$.

1. We can first apply F-SPLIT to split the top-level disjunction in τ . And suppose we want to work on the second disjunct first. This results in:

$$\tau_1 = (\text{len}(w) < 3 \wedge x \geq_{\text{str}} \text{"aabaa"}, \text{len}(w) \leq 3 \wedge x >_{\text{str}} \text{"aabaa"})$$

while the other elements of the state are unchanged.

2. Now, suppose we want to do binary search on the length objective. This can be done by again applying the F-SPLIT rule with the disjunction $(\text{len}(w) < 2 \wedge x \geq_{\text{str}} \text{"aabaa"}) \vee (2 \leq \text{len}(w) < 3 \wedge x \geq_{\text{str}} \text{"aabaa"})$ to get:

$$\begin{aligned} \tau_2 &= (\text{len}(w) < 2 \wedge x \geq_{\text{str}} \text{"aabaa"}, 2 \leq \text{len}(w) < 3 \wedge x \geq_{\text{str}} \text{"aabaa"}, \\ &\quad \text{len}(w) \leq 3 \wedge x >_{\text{str}} \text{"aabaa"}). \end{aligned}$$

3. Both F-SPLIT and F-SAT are applicable, but we follow the strategy of applying F-SAT after a split. Suppose we get the new solution $\mathcal{I}' = \{x \mapsto \text{"b"}, s \mapsto \epsilon, w \mapsto \text{"b"}\}$. Then we have:

$$\begin{aligned} \mathcal{I}_3 &= \{x \mapsto \text{"b"}, s \mapsto \epsilon, w \mapsto \text{"b"}\}, & \tau_3 &= (\Delta_3), \\ \Delta_3 &= (\text{len}(w) \leq 1 \wedge x >_{\text{str}} \text{"b"}) \vee (\text{len}(w) < 1 \wedge x \geq_{\text{str}} \text{"b"}). \end{aligned}$$

4. Both F-SPLIT and F-SAT are again applicable. Suppose that we switch now to linear search and thus again apply F-SAT, and suppose the new solution is $\mathcal{I}' = \{x \mapsto \text{"z"}, s \mapsto \epsilon, w \mapsto \text{"z"}\}$. This brings us to the state:

$$\begin{aligned} \mathcal{I}_4 &= \{x \mapsto \text{"z"}, s \mapsto \epsilon, w \mapsto \text{"z"}\}, & \tau_4 &= (\Delta_4), \\ \Delta_4 &= (\text{len}(w) \leq 1 \wedge x >_{\text{str}} \text{"z"}) \vee (\text{len}(w) < 1 \wedge x \geq_{\text{str}} \text{"z"}). \end{aligned}$$

5. Now, $\text{SOLVE}(\phi \wedge ((\text{len}(w) \leq 1 \wedge x >_{\text{str}} \text{"z"}) \vee (\text{len}(w) < 1 \wedge x \geq_{\text{str}} \text{"z"}))) = \perp$. Indeed, $\text{len}(w) \neq 0$, since $0 \leq \text{len}(s) < \text{len}(w)$; if $\text{len}(w) = 1$, then $\text{len}(s) = 0$ and $\text{len}(x) = 1$, thus, $x \not>_{\text{str}} \text{"z"}$. Now F-CLOSE can derive the state:

$$\langle \mathcal{I}_5, \Delta_5, \tau_5 \rangle = \langle \mathcal{I}_4, \Delta_4 \wedge \neg \Delta_4, \emptyset \rangle$$

6. We have reached a saturated state, and \mathcal{I}_5 is a Pareto optimal solution. \square

Optimization of objectives involving strings and integers (or strings and bitvectors) could be especially useful in security applications such as those mentioned in [60]. Optimization could be used in such applications to ensure that a counter-example is as simple as possible, for example.

Examples of multi-objective problems unsupported by existing solvers include multiple Pareto problems with a single min/max query, Pareto-lexicographic multi-objective optimization, and single Pareto queries involving MinMax and MaxMin optimization (see, for example, [1, 32, 52]). Our framework offers immediate solutions to these problems.

As has repeatedly been the case in SMT research, when new capabilities are introduced, new applications emerge. We expect that will happen also for the new capabilities introduced in this paper. One possible application is the optimization of emerging technology circuit designs [22].

4.4 Correctness

In this section, we establish correctness properties for \mathcal{GO} -derivations. Initially, we demonstrate that upon reaching a saturated state, the interpretation \mathcal{I} in that state is optimal.⁷

Theorem 1. (*Solution Soundness*) *Let $\langle \mathcal{I}, \Delta, \tau \rangle$ be a saturated state in a derivation for a GOMT problem \mathcal{GO} . Then, \mathcal{I} is an optimal solution to \mathcal{GO} .*

Proof. (Sketch) We show that in a saturated state $\tau = \emptyset$, and when $\tau = \emptyset$, $\phi \models \neg\Delta$. Then, we establish that \mathcal{I} is \mathcal{GO} -consistent, and that for any \mathcal{GO} -consistent \mathcal{T} -interpretation \mathcal{J} , $\mathcal{J} \models \Delta$ iff $\mathcal{J} <_{\mathcal{GO}} \mathcal{I}$. This implies there is no \mathcal{J} s.t. $\mathcal{J} \models \phi$ and $\mathcal{J} <_{\mathcal{GO}} \mathcal{I}$, confirming \mathcal{I} as an optimal solution to \mathcal{GO} . \square

In general, the calculus does not always have complete derivation strategies, for a variety of reasons. It could be that the problem is unbounded, i.e., no optimal solutions exist along some branch. Another possibility is that the order is not well-founded, and thus, an infinite sequence of improving solutions can be generated without ever reaching an optimal solution. For the former, various checks for unboundedness can be used. These are beyond the scope of this work, but some approaches are discussed in Trentin [64]. The latter can be overcome using a hybrid strategy when an optimization procedure exists (see Theorem 4). It is also worth observing that any derivation strategy is in effect an *anytime procedure*: forcibly stopping a derivation at any point yields (in the final state) the best solution found so far. When an optimal solution exists and is unique, stopping early provides the best approximation up to that point of the optimal solution.

There are also fairly general conditions under which solution complete derivation strategies do exist. We present them next.

⁷ Full proofs for the theorems in this section can be found in a longer version of this paper [67].

Definition 12. A derivation strategy is progressive if it (i) never halts in a non-saturated state and (ii) only uses F-SPLIT a finite number of times in any derivation.

Let us again fix a GOMT problem $\mathcal{GO} = \langle t, \prec, \phi \rangle$. Consider the set $A_t = \{t^{\mathcal{I}} \mid \mathcal{I} \text{ is } \mathcal{GO}\text{-consistent}\}$, collecting all values of t in interpretations satisfying ϕ .

Theorem 2. (Termination) If \prec is well-founded over A_t , any progressive strategy reaches a saturated state.

Proof. (Sketch) We show that any derivation using a progressive strategy terminates when \prec is well-founded. Subsequently, based on the definition of progressive, the final state must be saturated. \square

Theorem 3. (Solution Completeness) If \prec is well-founded over A_t and \mathcal{GO} has one or more optimal solutions, every derivation generated by a progressive derivation strategy ends with a saturated state containing one of them.

Proof. The proof is a direct consequence of Theorem 1 and Theorem 2. \square

Solution completeness can also be achieved using an appropriate hybrid strategy.

Theorem 4. If \mathcal{GO} has one or more optimal solutions and is not unbounded along any branch, then every derivation generated by a progressive hybrid strategy, where SOLVE is replaced by OPTIMIZE in F-SAT, ends with a saturated state containing one of them.

Proof. (Sketch) If D is such a derivation, we note that F-SPLIT can only be applied a finite number of times in D and consider the suffix of D after the last application of F-SPLIT. In that suffix, F-CLOSE can only be applied a finite number of times in a row, after which F-SAT must be applied. We then show that due to the properties of OPTIMIZE, this must be followed by either an application of F-CLOSE or a single application of F-SAT followed by F-CLOSE. Both cases result in saturated states. The theorem then follows from Theorem 1. \square

5 Related Work

Various approaches for solving the OMT problem have been proposed. We summarize the key ideas below and refer the reader to Trentin [64] for a more thorough survey.

The *offline schema* employs an SMT solver as a black box for optimization search through incremental calls [54, 55], following linear- or binary-search strategies. Initial bounds on the objective function are given and iteratively tightened after each call to the SMT solver. In contrast, the *inline schema* conducts the optimization search within the SMT solver itself [54, 55], integrating the optimization criteria into its internal algorithm. While the inline schema can be more efficient than the offline counterpart, it necessitates invasive changes to the solver and may not be possible for every theory.

Symbolic Optimization optimizes multiple independent linear arithmetic objectives simultaneously [36], seeking optimal solutions for each corresponding objective. This approach improves performance by sharing SMT search effort. It exists in both offline and inline versions, with the latter demonstrating superior performance. Other arithmetic schemas combine simplex, branch-and-bound, and cutting-plane techniques within SMT solvers [44, 50]. A polynomial constraint extension has also been introduced [33].

Theory-specific techniques address objectives involving pseudo-Booleans [11, 54, 55, 57], bitvectors [40, 65], bitvectors combined with floating-point arithmetic [66], and nonlinear arithmetic [6]. Other related work includes techniques for lexicographic optimization [8], Pareto optimization [8, 24], MaxSMT [19], and All-OMT [64].

Our calculus is designed to capture all of these variations. It directly corresponds to the offline schema, can handle both single- and multi-objective problems, and can integrate solvers with inline capabilities (including theory-specific ones) using the hybrid solving strategy. Efficient MaxSMT approaches [19] can also be mimicked in our calculus. These approaches systematically explore the search space by iteratively processing segments derived from unsat cores. Our calculus can instantiate these branches using the F-SPLIT rule, by first capturing unsat cores from calls to F-CLOSE, and then using these cores to direct the search in the F-SPLIT rule.

6 Conclusion and Future Work

This paper introduces the Generalized OMT problem, a proper extension of the OMT problem. It also provides a general setting for formalizing various approaches for solving the problem in terms of a novel calculus for GOMT and proves its key correctness properties. As with previous work on abstract transition systems for SMT [27, 31, 39, 45], this work establishes a framework for both theoretical exploration and practical implementations. The framework is general in several aspects: (i) it is parameterized by the optimization order, which does not need to be total; (ii) it unifies single- and multi-objective optimization problems in a single definition; (iii) it is theory-agnostic, making it applicable to any theory or combination of theories; and (iv) it provides a formal basis for understanding and exploring search strategies for Generalized OMT.

In future work, we plan to explore an extension of the calculus to the generalized All-OMT problem. We also plan to develop a concrete implementation of the calculus in a state-of-the-art SMT solver and evaluate it experimentally against current OMT solvers.

Acknowledgements. This work was funded in part by the Stanford Agile Hardware Center and by the National Science Foundation (grant 2006407).

References

1. Akinlana, D.M.: New Developments in Statistical Optimal Designs for Physical and Computer Experiments. Ph.D. thesis, University of South Florida (2022)
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org
3. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol.185, pp. 825–885. IOS Press (2009)
4. Bertolissi, C., dos Santos, D.R., Ranise, S.: Solving multi-objective workflow satisfiability problems with optimization modulo theories techniques. In: Bertino, E., Lin, D., Lobo, J. (eds.) Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018, pp. 117–128. ACM (2018)
5. Bian, Z., Chudak, F., Macready, W., Roy, A., Sebastiani, R., Varotti, S.: Solving SAT and MaxSAT with a quantum annealer: foundations and a preliminary report. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 153–171. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_9
6. Bigarella, F., et al.: Optimization modulo non-linear arithmetic via incremental linearization. In: Konev, B., Regehr, G. (eds.) FroCoS 2021. LNCS (LNAI), vol. 12941, pp. 213–231. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86205-3_12
7. Bit-Monnot, A., Leofante, F., Pulina, L., Abraham, E., Tacchella, A.: SMARtplan: a task planner for smart factories (2018). arXiv preprint [arXiv:1806.07135](https://arxiv.org/abs/1806.07135)
8. Bjørner, N.S., Phan, A.D.: νZ - maximal satisfaction with $z3$. In: International Symposium on Symbolic Computation in Software Science (2014)
9. Bjørner, N., Phan, A.-D., Fleckenstein, L.: νZ - An optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_14
10. Candeago, L., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Speeding up the constraint-based method in difference logic. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016, pp. 284–301. Springer International Publishing, Cham (2016)
11. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: foundations and applications. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 99–113. Springer, Berlin Heidelberg, Berlin, Heidelberg (2010)
12. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: A modular approach to MaxSAT modulo theories. In: Jarvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 150–165. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_12
13. Craciunas, S.S., Oliver, R.S., Chmelík, M., Steiner, W.: Scheduling real-time communication in IEEE 802.1qbv time sensitive networks. In: Proceedings of the 24th International Conference on Real-Time Networks and Systems, p. 183-192. RTNS 2016, Association for Computing Machinery, New York, NY, USA (2016)
14. Demarchi, S., Menapace, M., Tacchella, A.: Automating elevator design with satisfiability modulo theories. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI), pp. 26–33 (2019)
15. Demarchi, S., Tacchella, A., Menapace, M.: Automated design of complex systems with constraint programming techniques. In: Proceedings of the Cyber-Physical

- Systems Ph.D Workshop 2019, CPS Summer School “Designing Cyber-Physical Systems - From concepts to implementation”, Alghero, Italy, pp. 51–59 (2019)
16. Dutertre, B., de Moura, L.: Integrating simplex with DPPL(T). Technical Report, SRI International (2006)
 17. Eraşcu, M., Micota, F., Zaharie, D.: Applying optimization modulo theory, mathematical programming and symmetry breaking for automatic deployment in the cloud of component-based applications extended abstract. In: 4th Women in Logic Workshop, p. 6 (2020)
 18. Erata, F., Piskac, R., Mateu, V., Szefer, J.: Towards automated detection of single-trace side-channel vulnerabilities in constant-time cryptographic code (2023). arXiv preprint [arXiv:2304.02102](https://arxiv.org/abs/2304.02102)
 19. Fazekas, K., Bacchus, F., Biere, A.: Implicit hitting set algorithms for maximum satisfiability modulo theories. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 134–151. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_10
 20. Feng, J., Zhang, T., Yi, C.: Reliability-aware comprehensive routing and scheduling in time-sensitive networking. In: Wireless Algorithms, Systems, and Applications: 17th International Conference, WASA 2022, Dalian, China, November 24–26, 2022, Proceedings, Part II, pp. 243–254. Springer (2022)
 21. Gavran, I., Darulova, E., Majumdar, R.: Interactive synthesis of temporal specifications from examples and natural language. In: Proceedings of the ACM on Programming Languages, vol. 4(OOPSLA), pp. 1–26 (2020)
 22. Gretsch, R., Song, P., Madhavan, A., Lau, J., Sherwood, T.: Energy efficient convolutions with temporal arithmetic. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, pp. 354–368. ASPLOS 2024, Association for Computing Machinery, New York, NY, USA (2024)
 23. Henry, J., Asavaoae, M., Monniaux, D., Maïza, C.: How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In: Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, Edinburgh, United Kingdom, pp. 43–52. Association for Computing Machinery, New York, NY, USA (2014)
 24. Jackson, D., Estler, H.C., Rayside, D.: The Guided Improvement Algorithm for Exact, General-Purpose, Many-Objective Combinatorial Optimization. Technical Report 2009-033, MIT-CSAIL (2009)
 25. Jiang, J., Chen, L., Wu, X., Wang, J.: Block-wise abstract interpretation by combining abstract domains with SMT. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10145, pp. 310–329. Springer (2017). https://doi.org/10.1007/978-3-319-52234-0_17
 26. Jin, X., Xia, C., Guan, N., Zeng, P.: Joint algorithm of message fragmentation and no-wait scheduling for time-sensitive networks. IEEE/CAA J. Automatica Sin. **8**(2), 478–490 (2021)
 27. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
 28. Karpenkov, G.E.: Finding inductive invariants using satisfiability modulo theories and convex optimization, Ph.D Thesis. Université Grenoble Alpes (2017). https://tel.archives-ouvertes.fr/tel-01681555/file/KARPENKOV_2017_diffusion.pdf

29. Knüsel, M.: Optimizing Declarative Power Sequencing. Master thesis, ETH Zurich, Zurich (2021-09)
30. Kovásznai, G., Biró, C., Erdélyi, B.: Puli - a problem-specific OMT solver. Easy-Chair Preprints (2018)
31. Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCoS 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74621-8_1
32. Lai, L., Fiaschi, L., Cococcioni, M., Deb, K.: Pure and mixed lexicographic-pretian many-objective optimization: state of the art. *Nat. Comput. Int. J.* **22**(2), 227–242 (2022)
33. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Minimal-model-guided approaches to solving polynomial constraints and extensions. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014, pp. 333–350. Springer International Publishing, Cham (2014)
34. Lee, D., et al.: SP&R: SMT-based simultaneous place-and-route for standard cell synthesis of advanced nodes. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **40**(10), 2142–2155 (2020)
35. Leofante, F., Giunchiglia, E., Ábrahám, E., Tacchella, A.: Optimal planning modulo theories. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, pp. 4128–4134 (2021)
36. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, USA, pp. 607–618 (2014). <https://doi.org/10.1145/2535838.2535857>
37. Liu, T., Tyszbrowicz, S., Beckert, B., Taghdiri, M.: Computing exact loop bounds for bounded program verification. In: Larsen, K.G., Sokolsky, O., Wang, J. (eds.) SETTA 2017. LNCS, vol. 10606, pp. 147–163. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69483-2_9
38. Marchetto, G., Sisto, R., Valenza, F., Yusupov, J., Ksentini, A.: A formal approach to verify connectivity and optimize VNF placement in industrial networks. *IEEE Trans. Industr. Inf.* **17**(2), 1515–1525 (2021)
39. de Moura, L.M., Bjørner, N.S.: Model-based theory combination. In: Krstić, S., Oliveras, A. (eds.) Proceedings of the 5th International Workshop on Satisfiability Modulo Theories, SMT@CAV 2007, Berlin, Germany, July 1-2, 2007. Electronic Notes in Theoretical Computer Science, vol. 198, pp. 37–49. Elsevier (2007)
40. Nadel, A., Rychin, V.: Bit-vector optimization. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 851–867. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_53
41. Nguyen, C.M., Sebastiani, R., Giorgini, P., Mylopoulos, J.: Requirements evolution and evolution requirements with constrained goal models. In: Comyn-Wattiau, I., Tanaka, K., Song, I.-Y., Yamamoto, S., Saeki, M. (eds.) ER 2016. LNCS, vol. 9974, pp. 544–552. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46397-1_42
42. Nguyen, C.M., Sebastiani, R., Giorgini, P., Mylopoulos, J.: Modeling and reasoning on requirements evolution with constrained goal models. In: Cimatti, A., Sirjani, M. (eds.) Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10469, pp. 70–86. Springer (2017)

43. Nguyen, C.M., Sebastiani, R., Giorgini, P., Mylopoulos, J.: Multi-objective reasoning with constrained goal models. *Requirements Eng.* **23**(2), 189–225 (2016). <https://doi.org/10.1007/s00766-016-0263-5>
44. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) *Theory and Applications of Satisfiability Testing - SAT 2006*, pp. 156–169. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)
45. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. ACM* **53**(6), 937–977 (2006)
46. Paoletti, N., et al.: Synthesizing stealthy reprogramming attacks on cardiac devices. In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019*, pp. 13–22. Association for Computing Machinery, New York, NY, USA (2019)
47. Park, D., Lee, D., Kang, I., Gao, S., Lin, B., Cheng, C.K.: SP&R: simultaneous placement and routing framework for standard cell synthesis in sub-7nm. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 345–350 (2020)
48. Patti, G., Bello, L.L., Leonardi, L.: Deadline-aware online scheduling of tsn flows for automotive applications. *IEEE Trans. Ind. Inform.* **19**(4), 5774–5784 (2022)
49. Ratschan, S.: Simulation based computation of certificates for safety of dynamical systems (2017). arXiv preprint [arXiv:1707.00879](https://arxiv.org/abs/1707.00879)
50. Roc, O.V.: Optimization Modulo Theories. Master’s thesis, Polytechnic University of Catalonia. <https://upcommons.upc.edu/handle/2099.1/14204> (2011)
51. Rybalchenko, A., Vuppalapati, C.: Supercharging plant configurations using z3. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings*. vol. 12735, p. 1. Springer Nature (2021)
52. Schmitt, T., Hoffmann, M., Rodemann, T., Adamy, J.: Incorporating human preferences in decision making for dynamic multi-objective optimization in model predictive control. *Inventions* **7**(3), 46 (2022)
53. Sebastiani, R., Trentin, P.: Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In: Baier, C., Tinelli, C. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 335–349. Springer, Berlin Heidelberg, Berlin, Heidelberg (2015)
54. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{L}\mathcal{A}(\mathbb{Q})$ cost functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) *Automated Reasoning*, pp. 484–498. Springer, Berlin Heidelberg, Berlin, Heidelberg (2012)
55. Sebastiani, R., Tomasi, S.: Optimization modulo theories with linear rational costs. *ACM Trans. Comput. Logic* **16**(2), 1–43 (2015)
56. Sebastiani, R., Trentin, P.: OptiMathSAT: a tool for optimization modulo theories. *J. Autom. Reasoning* **64**(3), 423–460 (2018). <https://doi.org/10.1007/s10817-018-09508-6>
57. Sebastiani, R., Trentin, P.: On optimization modulo theories, maxSMT and sorting networks. In: Legay, A., Margaria, T. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 231–248. Springer, Berlin Heidelberg, Berlin, Heidelberg (2017)
58. Shen, D., Zhang, T., Wang, J., Deng, Q., Han, S., Hu, X.S.: QoS guaranteed resource allocation for coexisting eMBB and URLLC traffic in 5G industrial networks. In: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 81–90. IEEE (2022)

59. Sivaraman, A., Farnadi, G., Millstein, T., Van den Broeck, G.: Counterexample-guided learning of monotonic neural networks. *Adv. Neural. Inf. Process. Syst.* **33**, 11936–11948 (2020)
60. Subramanian, S., Berzish, M., Tripp, O., Ganesh, V.: A solver for a theory of strings and bit-vectors. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 124–126 (2017)
61. Tarrach, T., Ebrahimi, M., König, S., Schmittner, C., Bloem, R., Nickovic, D.: Thorsten Tarrach and Masoud Ebrahimi and Sandra König and Christoph Schmittner and Roderick Bloem and Dejan Nickovic, WorkingPaper (2022)
62. Teso, S., Sebastiani, R., Passerini, A.: Structured learning modulo theories. *Artif. Intell.* **244**, 166–187 (2017)
63. Tierno, A., Turri, G., Cimatti, A., Passerone, R.: Symbolic encoding of reliability for the design of redundant architectures. In: 2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS), pp. 01–06 (2022)
64. Trentin, P.: Optimization Modulo Theories with OptiMathSAT. Ph.D. thesis, University of Trento (2019)
65. Trentin, P., Sebastiani, R.: Optimization modulo the theories of signed bit-vectors and floating-point numbers. *J. Autom. Reason.* **65**(7), 1071–1096 (2021)
66. Trentin, P., Sebastiani, R.: Optimization modulo the theories of signed bit-vectors and floating-point numbers. *J. Autom. Reason.* **65**(7), 1071–1096 (2021)
67. Tsiskaridze, N., Barrett, C., Tinelli, C.: Generalized optimization modulo theories (2024). arXiv preprint [arXiv:2404.16122](https://arxiv.org/abs/2404.16122)
68. Tsiskaridze, N., et al.: Automating system configuration. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19–22, 2021, pp. 102–111. IEEE (2021)
69. Yao, P., Shi, Q., Huang, H., Zhang, C.: Program analysis via efficient symbolic abstraction. In: Proceedings of the ACM on Programming Languages, vol. 5(OOP-SLA), pp. 1–32 (2021)

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

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



Author Index

A

Acclavio, Matteo II-216
Amrollahi, Daneshvar I-154
Arrial, Victor II-338
Avigad, Jeremy I-3
Ayala-Rincón, Mauricio II-317

B

Baader, Franz II-279
Balbiani, Philippe II-78
Barragán, Andrés Felipe González II-317
Barrett, Clark I-458
Bártek, Filip I-194
Berg, Jeremias I-396
Bhayat, Ahmed I-75
Biere, Armin I-284
Bonsangue, Marcello II-401
Bozec, Tanguy II-157
Bromberger, Martin I-133
Brown, Chad E. I-86
Bruni, Alessandro II-61

C

Cerna, David M. II-317
Chassot, Samuel I-304
Chvalovský, Karel I-194
Ciabattini, Agata II-176
Coopmans, Tim II-401

D

Das, Anupam II-237
De Lon, Adrian I-105
De, Abhishek II-237
Dixon, Clare II-3

E

Ehling, Georg II-381
Einarsdóttir, Sólrún Halla I-214

F

Férée, Hugo II-43
Fernández Gil, Oliver II-279
Ferrari, Mauro II-24
Fiorentini, Camillo II-24
Frohn, Florian I-344
Froleyks, Nils I-284
Fruzsa, Krisztina II-114

G

Gao, Han II-78
Garcia, Ronald I-419
Ge, Rui I-419
Gencer, Çiğdem II-78
Ghilardi, Silvio I-265
Giesl, Jürgen I-233, I-344, II-360
Giessen, Iris van der II-43
Gool, Sam van II-43
Graham-Lengrand, Stéphane I-386
Guerrieri, Giulio II-338

H

Hader, Thomas I-386
Hajdu, Márton I-21, I-115, I-154, I-214
Heisinger, Maximilian I-315, I-325
Heisinger, Simone I-315, I-325
Heljanko, Keijo I-284
Heuer, Jan I-172
Hozzová, Petra I-21, I-154
Hustadt, Ullrich II-3

I

Ihalainen, Hannes I-396
Irfan, Ahmed I-386

J

Järvisalo, Matti I-396
Johansson, Moa I-214

K

Kaliszyk, Cezary I-86
 Kassing, Jan-Christoph II-360
 Kaufmann, Daniela I-386
 Kesner, Delia II-338
 Khalid, Zain I-53
 Kotthoff, Lars I-53
 Kovács, Laura I-21, I-115, I-154, I-386
 Kozen, Dexter II-257
 Krasnopol, Florent I-133
 Kunčák, Viktor I-304
 Kutsia, Temur II-317, II-381
 Kuznets, Roman II-114

L

Laarman, Alfons II-401
 Lammich, Peter I-439
 Lommen, Nils I-233

M

Mei, Jingyi II-401
 Meyer, Éléanore I-233
 Middeldorp, Aart II-298
 Mitterwallner, Fabian II-298
 Möhle, Sibylle I-133
 Myreen, Magnus O. I-396

N

Nalon, Cláudia II-3, II-97
 Niederhauser, Johannes I-86
 Nordström, Jakob I-396

O

Oertel, Andy I-396
 Olivetti, Nicola II-78

P

Papacchini, Fabio II-3
 Pattinson, Dirk II-97
 Peltier, Nicolas II-157
 Perrault, C. Raymond I-53
 Petitjean, Quentin II-157
 Platzer, André II-196

Poidomani, Lia M. I-265
 Pommellet, Adrien I-366
 Prebet, Enguerrand II-196

R

Rawson, Michael I-115
 Rebola-Pardo, Adrian I-325
 Ritter, Eike II-61
 Rooduijn, Jan II-257
 Ruess, Harald II-137

S

Scatton, Simon I-366
 Schmid, Ulrich II-114
 Schöpf, Jonas II-298
 Schürmann, Carsten II-61
 Seidl, Martina I-315, I-325
 Shillito, Ian II-43
 Sighireanu, Mihaela II-157
 Silva, Alexandra II-257
 Smallbone, Nicholas I-214
 Stan, Daniel I-366
 Suda, Martin I-75, I-194, I-214
 Summers, Alexander J. I-419
 Sutcliffe, Geoff I-30, I-53
 Suttner, Christian I-53

T

Tan, Yong Kiam I-396
 Tesi, Matteo II-176
 Tinelli, Cesare I-458
 Tsiskaridze, Nestan I-458

V

van Ditmarsch, Hans II-114
 Vartanyan, Grigory II-360
 Voronkov, Andrei I-21, I-115, I-154

W

Wagner, Eva Maria I-154
 Waldmann, Uwe I-244
 Weidenbach, Christoph I-133
 Wernhard, Christoph I-172

Y

Yu, Emily I-284