Uli Sattler
Martin Suda (Eds.)

LNAI 14279

Frontiers of Combining Systems

14th International Symposium, FroCoS 2023
Prague, Czech Republic, September 20–22, 2023
Proceedings

OPEN ACCESS

Lecture Notes in Computer Science

Lecture Notes in Artificial Intelligence 14279

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.

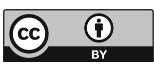
Uli Sattler · Martin Suda
Editors

Frontiers of Combining Systems

14th International Symposium, FroCoS 2023
Prague, Czech Republic, September 20–22, 2023
Proceedings

Editors
Uli Sattler
University of Manchester
Manchester, UK

Martin Suda
Czech Technical University in Prague
Prague, Czech Republic



ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Artificial Intelligence
ISBN 978-3-031-43368-9 ISBN 978-3-031-43369-6 (eBook)
https://doi.org/10.1007/978-3-031-43369-6

LNCS Sublibrary: SL7 – Artificial Intelligence

© The Editor(s) (if applicable) and The Author(s) 2023. 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

Paper in this product is recyclable.

Preface

These proceedings contain the papers selected for presentation at the 14th *International Symposium on Frontiers of Combining Systems* (FroCoS 2023). The symposium was held during September 20–22, 2023 at Czech Technical University in Prague (CTU), Czech Republic. It was co-located with the 32nd *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (TABLEAUX 2023).

FroCoS is the main international event for research on the development of techniques and methods for the combination and integration of formal systems, their modularization and analysis. Previous FroCoS meetings have been organized across the world, since 1996; see Figures 1 and 2 for a global and a European view of the locations of past and present meetings.

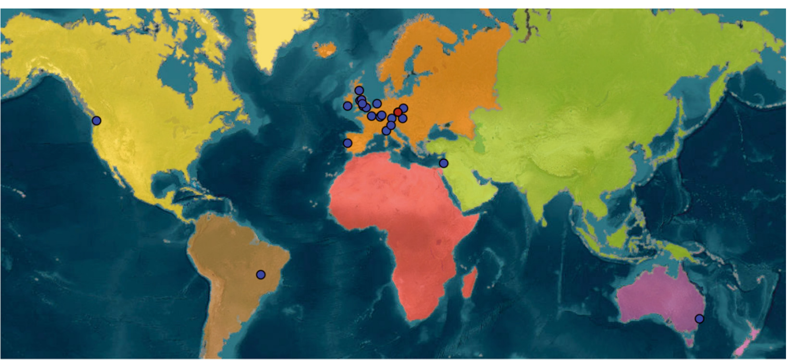


Fig. 1. A global map showing locations of past and current FroCoS meetings

FroCoS 2023 received 22 high-quality paper submissions, which were evaluated by the members of the Program Committee who did a great job at thoroughly evaluating these submissions regarding their technical and presentational quality and providing helpful feedback to the authors. Reviewing was single-blind and each paper was subject to at least three reviews, followed by sometimes extensive discussions within the Program Committee and, in three cases, a second round of reviewing. In the end, 14 papers were selected for presentation at the symposium and for publication. We have grouped them in this volume according to the following topic classification: (1) analysis of programs and equations, (2) unification, (3) decidable fragments, (4) frameworks, and (5) higher-order theorem proving.

Together with the Program Committee, we considered suitable candidates to give an invited talk, and were delighted to have found five outstanding invited speakers:

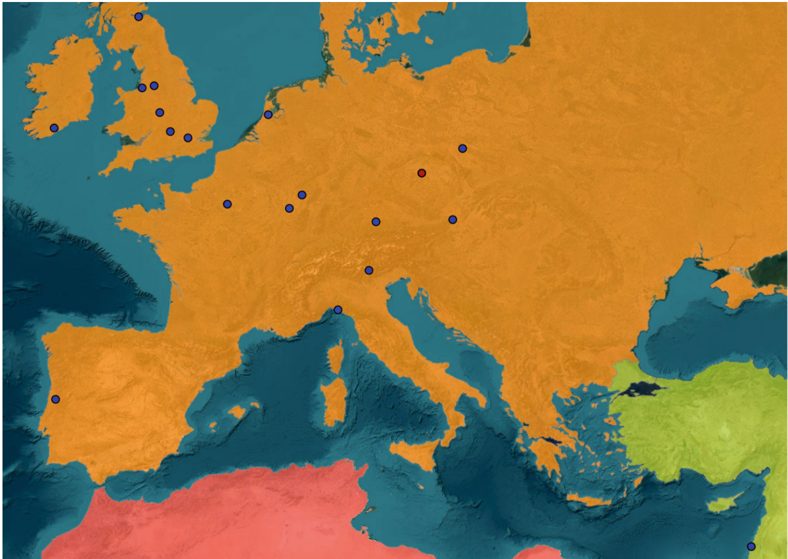


Fig. 2. A Europe-centric map showing locations of past and current FroCoS meetings in Europe and Asia

- Marta Bílková, Czech Academy of Sciences, Czechia (joint with TABLEAUX 2023)
- Chad E. Brown, Czech Technical University in Prague, Czechia (joint with TABLEAUX 2023)
- Valentin Goranko, Stockholm University, Sweden (joint with TABLEAUX 2023)
- Katalin Fazekas, TU Wien, Austria
- Yoni Zohar, Bar-Ilan University, Israel

We would like to thank all the people who contributed to making FroCoS 2023 a success. In particular, we thank the members of the Program Committee and the external reviewers for their excellent, timely work and for providing the authors with insightful feedback. Of course we thank the authors for submitting high-quality papers, taking the reviewers' feedback into account, and presenting their work in a way that is accessible to the broad FroCoS audience. Next, we thank the invited speakers for their inspiring talks. Moreover, we thank the local organisers and the Czech Technical University in Prague for organising and supporting FroCoS. Finally, we gratefully acknowledge financial support from Springer.

July 2023

Uli Sattler
Martin Suda

Organization

Program Committee Chairs

Uli Sattler	University of Manchester, UK
Martin Suda	Czech Technical University in Prague, Czech Republic

Steering Committee

Franz Baader	Dresden University of Technology, Germany
Clare Dixon	University of Manchester, UK
Marcelo Finger	University of São Paulo, Brazil
Andreas Herzig	Université Paul Sabatier in Toulouse, France
Boris Konev	University of Liverpool, UK
Andrei Popescu	University of Sheffield, UK
Giles Reger	Amazon Web Services, USA & University of Manchester, UK

Program Committee

Carlos Areces	Universidad Nacional de Córdoba, Argentina
Alessandro Artale	Free University of Bolzano-Bozen, Italy
Franz Baader	TU Dresden, Germany
Haniel Barbosa	Universidade Federal de Minas Gerais, Brazil
Peter Baumgartner	CSIRO Canberra, Australia
Clare Dixon	University of Manchester, UK
Mathias Fleury	University of Freiburg, Germany
Didier Galmiche	LORIA, Université de Lorraine, France
Silvio Ghilardi	Università degli Studi di Milano, Italy
Jürgen Giesl	RWTH Aachen University, Germany
Andreas Herzig	IRIT at Université Paul Sabatier, France
Roman Kontchakov	Birkbeck, University of London, UK
Paliath Narendran	University at Albany - SUNY, USA
Aina Niemetz	Stanford University, USA
Naoki Nishida	Nagoya University, Japan

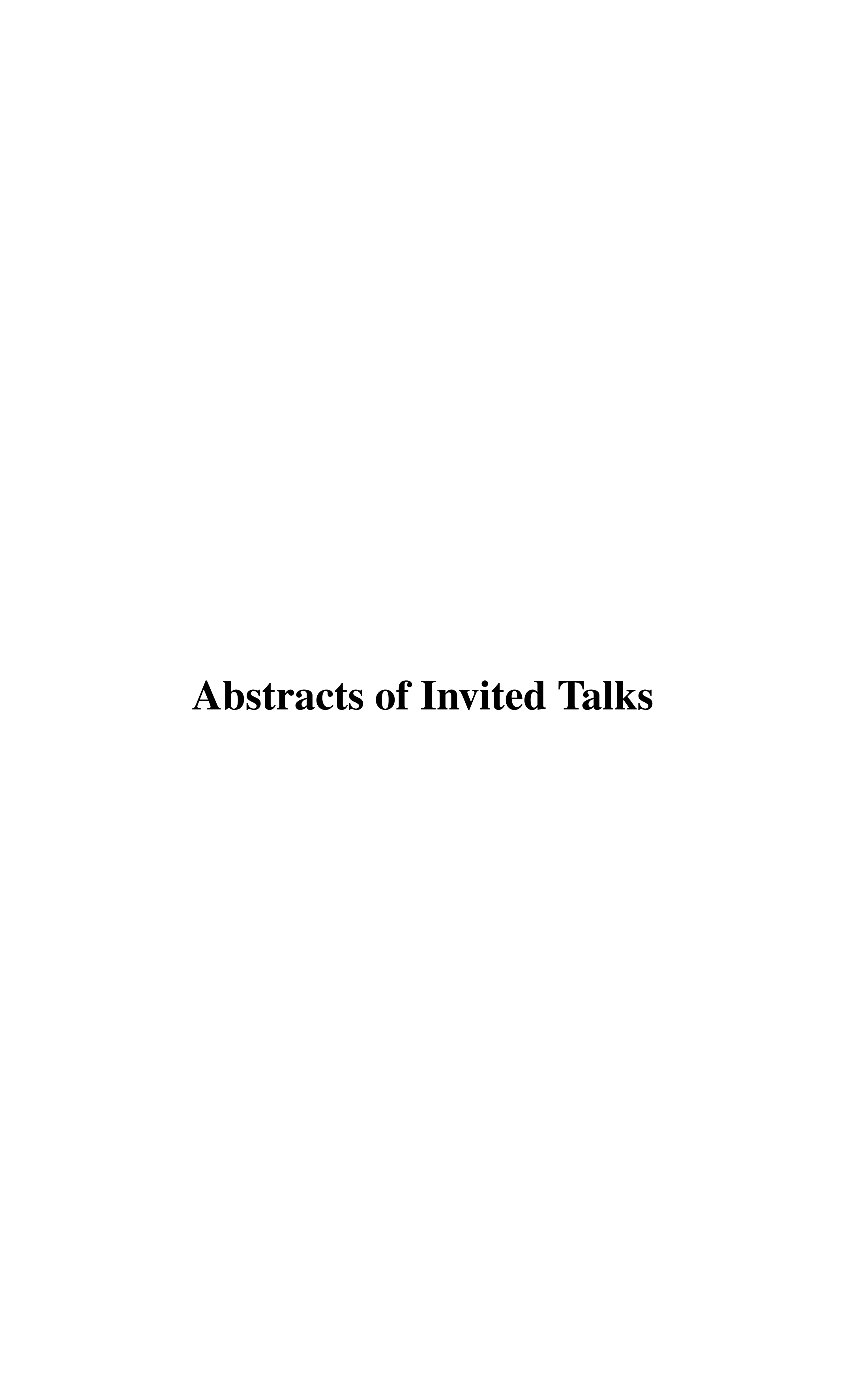
Giles Reger	Amazon Web Services, USA & University of Manchester, UK
Andrew Reynolds	University of Iowa, USA
Christophe Ringeissen	LORIA, Université de Lorraine, France
Philipp Rümmer	University of Regensburg, Germany
Renate A. Schmidt	University of Manchester, UK
Roberto Sebastiani	University of Trento, Italy
Viorica Sofronie-Stokkermans	University of Koblenz, Germany
K. Subramani	West Virginia University, USA
Dmitriy Traytel	University of Copenhagen, Denmark
Christoph Weidenbach	Max Planck Institute for Informatics, Germany
Piotr Wojciechowski	West Virginia University, USA
Akihisa Yamada	AIST, Japan

Local Organisers

Karel Chvalovský	Czech Technical University in Prague, Czech Republic
Jan Jakubův	Czech Technical University in Prague, Czech Republic
Martin Suda	Czech Technical University in Prague, Czech Republic
Josef Urban	Czech Technical University in Prague, Czech Republic
Cezary Kaliszyk	University of Innsbruck, Austria

Additional Reviewers

Daniel Cloerkes	RWTH Aachen University, Germany
Jan-Christoph Kassing	RWTH Aachen University, Germany
Nao Hirokawa	JAIST, Japan
Hans-Jörg Schurr	University of Iowa, USA
Karel Chvalovský	Czech Technical University in Prague, Czech Republic
Boris Konev	Liverpool University, UK
Madalina Erascu	West University of Timisoara, Romania



Abstracts of Invited Talks

Incremental Reasoning in Embedded SAT Solvers

Katalin Fazekas

TU Wien, Austria

Abstract. Embedding SAT solvers as sub-reasoning engines into more complex tools is a common practice in various application domains. For instance, SAT-based model checkers exploit modern solvers as black-box oracles, while solvers for Satisfiability Modulo Theories (SMT), Maximum Satisfiability (MaxSAT) or other combinatorial problems combine SAT solvers with various reasoning or optimization engines. Such embedded SAT solvers are used incrementally in most cases, i.e., the exact same SAT solver instance is reused to solve multiple related SAT queries. The goal of incremental reasoning is to exploit the shared constraints between consecutive SAT queries and thereby avoid repeated work and reduce solving time.

In this talk, first we briefly survey the functionalities supported by IPASIR, the standard API of incremental SAT solvers, which integrates solvers as black-boxes into larger systems. Then, we present our recently proposed extension to that interface which allows us to modify and refine SAT queries already during solving and thereby to benefit from incremental reasoning even more. The proposed extension, as we demonstrate by our experiments, captures the most essential functionalities that are sufficient to simplify and improve use cases where a more fine-grained interaction between the SAT solver and the rest of the system is required. We will present our experiments where we extended CaDiCaL, a state-of-the-art incremental SAT solver, with our proposed interface and evaluated it on two representative use cases: enumerating graphs within the SAT modulo Symmetries framework (SMS), and embedding it as the main CDCL(T) SAT engine in the SMT solver cvc5. Following that, we overview the key open challenges in such use cases to efficiently combine some complex crucial features of modern SAT solvers, such as inprocessing and proof production, with incremental reasoning. At the end, we briefly present possible ways to address some of these challenges.

This is a joint work with Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere.

On Datatypes, Synergies, and Unicorns: Recent Developments in Theory Combination

Yoni Zohar

Bar-Ilan University, Israel

Abstract. A Satisfiability Modulo Theories (SMT) solver is a tool that takes as input a first-order formula, and determines its T-satisfiability, that is, the existence of a first-order structure that satisfies it, as well as the axioms of some first-order theory T. Some theories are considered primitive, such as the theories of integers, reals, arrays, and lists. Other theories are considered combined, as they are obtained by the combination of existing theories. Examples include the theory of arrays of integers, or of lists of reals.

Now, assume that you have an SMT solver that supports two theories. How hard would it be to extend it so that it supports their combination? The classical answer to this question was given by Nelson-Oppen. They designed a decision procedure for a given combined theory by first purifying the input formula to two parts, one for each theory; then guessing equalities and disequalities between the shared variables of the two parts; and finally calling the two decision procedures for the separate theories on the part of the purified formula that is relevant to them, plus the guessed set of (dis)equalities.

The correctness of this combination method requires the two combined theories to be stably infinite, a model theoretic property related to the existence of infinite models. However, not all theories of interest are stably infinite. (For example, the theory of fixed-size bit-vectors is not.)

This state of affairs led to the development of various other combination methods that rely on various model theoretic notions, such as shiny, gentle, and polite theories. For each combination method, the corresponding properties of the theories need to be proven in order to be used with that method. And indeed, various theories have been shown to admit such properties.

In this talk I will survey recent results in the field of theory combination. First, I will sketch a proof that theories of datatypes (e.g., lists, trees) can be combined with any other theory, using the polite combination method. Next, I will show how the original Nelson-Oppen method

can be integrated together with the polite combination method in a synergetic way that reduces the number of guesses one needs to make. Finally, a taxonomy of various model theoretic properties from theory combination will be presented, where the properties will be analyzed and compared. This will include the description of open problems which relate to a certain kind of theories (that are called “unicorns”).

Contents

Analysis of Programs and Equations

Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs . . . 3
Nils Lommen and Jürgen Giesl

Recurrence-Driven Summations in Automated Deduction . . . 23
Visa Nummelin, Jasmin Blanchette, and Sander R. Dahmen

Formal Verification of Bit-Vector Invertibility Conditions in Coq . . . 41
Burak Ekici, Arjun Viswanathan, Yoni Zohar, Cesare Tinelli, and Clark Barrett

Unification

Weighted Path Orders Are Semantic Path Orders . . . 63
Teppei Saito and Nao Hirokawa

KBO Constraint Solving Revisited . . . 81
Yasmine Briefs, Hendrik Leidinger, and Christoph Weidenbach

A Critical Pair Criterion for Level-Commutation of Conditional Term Rewriting Systems . . . 99
Ryota Haga, Yuki Kagaya, and Takahito Aoto

Decidable Fragments

Logic of Communication Interpretation: How to Not Get Lost in Translation . . . 119
Giorgio Cignarale, Roman Kuznets, Hugo Rincon Galeana, and Ulrich Schmid

Symbolic Model Construction for Saturated Constrained Horn Clauses . . . 137
Martin Bromberger, Lorenz Leutgeb, and Christoph Weidenbach

Frameworks

Combining Finite Combination Properties: Finite Models and Busy Beavers . . . 159
Guilherme V. Toledo, Yoni Zohar, and Clark Barrett

Formal Reasoning Using Distributed Assertions 176
Farah Al Wardani, Kaustuv Chaudhuri, and Dale Miller

An Abstract CNF-to-d-DNNF Compiler Based on Chronological CDCL 195
Sibylle Möhle

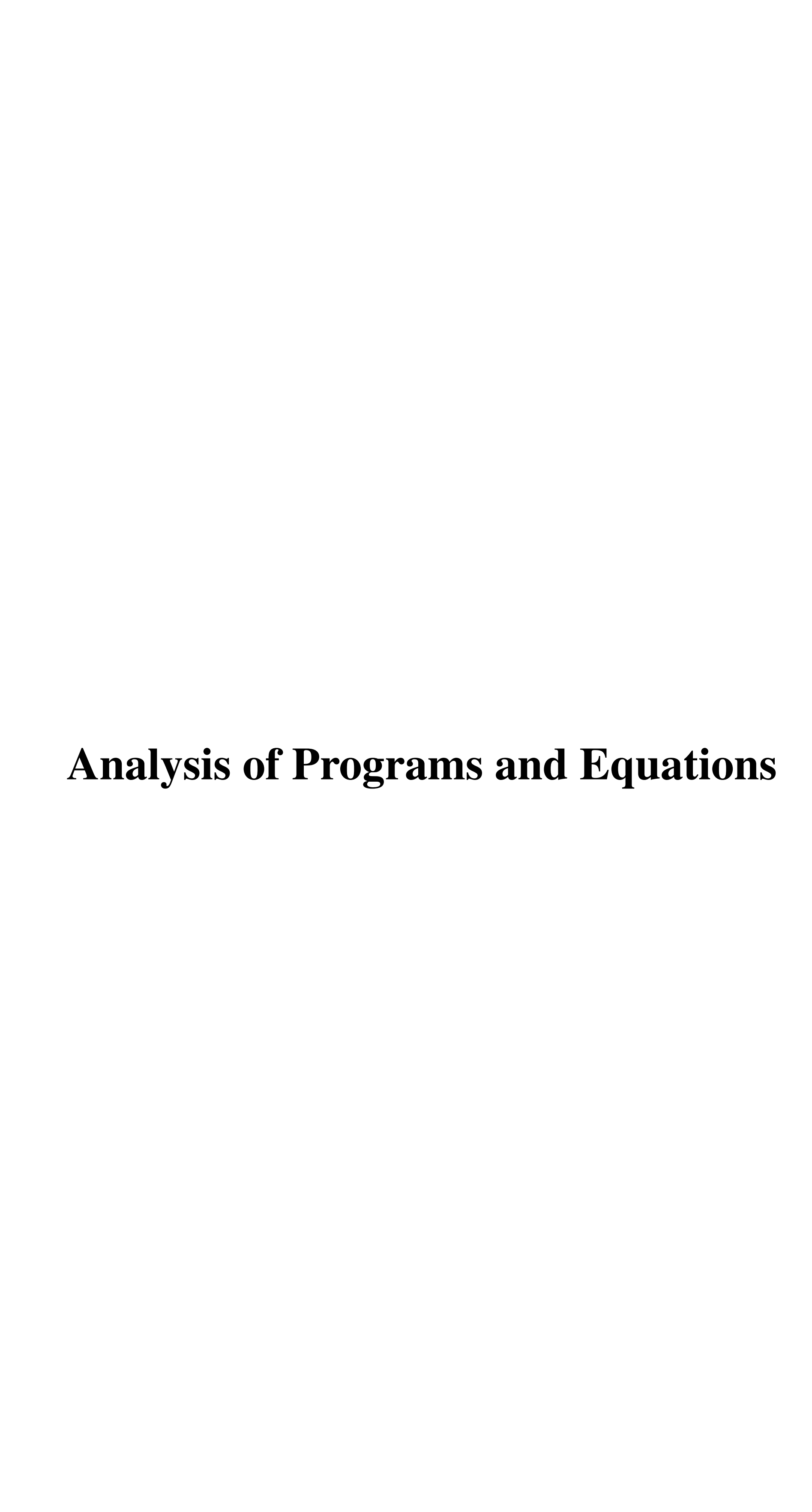
Higher-Order Theorem Proving

Hammering Floating-Point Arithmetic 217
Olle Torstensson and Tjark Weber

Learning Proof Transformations and Its Applications in Interactive Theorem Proving 236
Liao Zhang, Lasse Blaauwbroek, Cezary Kaliszyk, and Josef Urban

Translating SUMO-K to Higher-Order Set Theory 255
Chad E. Brown, Adam Pease, and Josef Urban

Author Index 275



Analysis of Programs and Equations

Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs

Nils Lommen(✉) and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
lommen@cs.rwth-aachen.de, giesl@informatik.rwth-aachen.de

Abstract. We present a new procedure to infer *size bounds* for integer programs automatically. Size bounds are important for the deduction of bounds on the runtime complexity or in general, for the resource analysis of programs. We show that our technique is *complete* (i.e., it always computes finite size bounds) for a subclass of loops, possibly with non-linear arithmetic. Moreover, we present a novel approach to combine and integrate this complete technique into an incomplete approach to infer size and runtime bounds of general integer programs. We prove completeness of our integration for an important subclass of integer programs. We implemented our new algorithm in the automated complexity analysis tool KoAT to evaluate its power, in particular on programs with non-linear arithmetic.

1 Introduction

There are numerous incomplete approaches for automatic resource analysis of programs, e.g., [1,2,5,8,10,15,19,21,29,33]. However, also many complete techniques to decide termination, analyze runtime complexity, or study memory consumption for certain classes of programs have been developed, e.g., [3,4,6,7,16,17,20,22,27,34,36]. In this paper, we present a procedure to compute *size bounds* which indicate how large the absolute value of an integer variable may become. In contrast to other complete procedures for the inference of size bounds which are based on fixpoint computations [3,6], our technique can also handle (possibly negative) constants and exponential size bounds. Similar to our earlier paper [27], we embed a procedure which is *complete* for a subclass of loops (i.e., it computes finite size bounds for all loops from this subclass) into an incomplete approach for general integer programs [8,19]. In this way, the power of the incomplete approach is increased significantly, in particular for programs with non-linear arithmetic. However, in the current paper we tackle a completely different problem than in [27] (and thus, the actual new contributions are also completely different), because in [27] we embedded a complete technique in order

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2).

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 3–22, 2023.
https://doi.org/10.1007/978-3-031-43369-6_1

to infer runtime bounds, whereas now we integrate a novel technique in order to infer size bounds. As an example, we want to determine bounds on the absolute values of the variables during (and after) the execution of the following loop.

$$\textbf{while } (x_3 > 0) \textbf{ do } (x_1, x_2, x_3, x_4) \leftarrow (3 \cdot x_1 + 2 \cdot x_2, -5 \cdot x_1 - 3 \cdot x_2, x_3 - 1, x_4 + x_3^2) \quad (1)$$

We introduce a technique to compute size bounds for loops which admit a closed form, i.e., an expression which corresponds to applying the loop's update n times. Then we over-approximate the closed form to obtain a non-negative, weakly monotonically increasing function. For instance, a closed form for x_3 in our example is $x_3 - n$, since the value of x_3 is decreased by n after n iterations. The (absolute value of this) closed form can be over-approximated by $x_3 + n$, which is monotonically increasing in all variables. Finally, each occurrence of n is substituted by a runtime bound for the loop. Clearly, (1) terminates after at most x_3 iterations. So if we substitute n by the runtime bound x_3 in the over-approximated closed form $x_3 + n$, then we infer the linear bound $2 \cdot x_3$ on the size of x_3. Due to the restriction to weakly monotonically increasing over-approximations, we can plug in any over-approximation of the runtime and do not necessarily need exact bounds.

Structure. We introduce our technique to compute size bounds by closed forms in Sect. 2 and show that it is complete for a subclass of loops in Sect. 3. Afterwards in Sect. 4, we incorporate our novel technique into the incomplete setting of general integer programs. In Sect. 5 we demonstrate how size bounds are used in automatic complexity analysis and study completeness for classes of general programs. In Sect. 6, we conclude with an experimental evaluation of our implementation in the tool KoAT and discuss related work. All proofs can be found in [28].

2 Size Bounds by Closed Forms

In this section, we present our novel technique to compute size bounds for loops by closed forms in Theorem 7. We start by introducing the required preliminaries. Let $\mathcal{V} = \{x_1, \ldots, x_d\}$ be a set of variables. $\mathcal{F}(\mathcal{V})$ is the set of all *formulas* built from inequations $p > 0$ for polynomials $p \in \mathbb{Q}[\mathcal{V}]$, $\wedge$, and $\vee$. A *loop* (φ, η) consists of a guard $\varphi \in \mathcal{F}(\mathcal{V})$ and an update $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ mapping variables to polynomials. A *closed form* $\mathtt{cl}^{x_i}$ (formally defined in Definition 1 below) is an expression in n and in the (initial values of the) variables $x_1, \ldots, x_d$ which corresponds to the value of x_i after iterating the loop n times. For our purpose we only need closed forms which hold for all $n \geq n_0$ for some fixed $n_0 \in \mathbb{N}$. Moreover, we restrict ourselves to closed forms which are so-called normalized poly-exponential expressions [16]. Nonetheless, our procedure works for any closed form expression with a finite number of arithmetic operations (i.e., the number of operations must be independent of n). We extend the application of functions like $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ also to polynomials, vectors, and formulas, etc., by replacing each variable v in the expression by $\eta(v)$. So in particular, $(\eta_2 \circ \eta_1)(x) = \eta_2(\eta_1(x))$

stands for the polynomial $\eta_1(x)$ in which every variable v is replaced by $\eta_2(v)$. Moreover, η^n denotes the n-fold application of η.

We call a function $\sigma : \mathcal{V} \to \mathbb{Z}$ a *state*. By $\sigma(exp)$ or $\sigma(\varphi)$ we denote the number resp. Boolean value which results from replacing every variable v by the number $\sigma(v)$ in the arithmetic expression *exp* or the formula φ.

Definition 1 (Closed Forms). *For a loop (φ, η), an arithmetic expression $\texttt{cl}^{x_i}$ is a* closed form *for x_i with* start value *$n_0 \in \mathbb{N}$ if $\texttt{cl}^{x_i} = \sum_{1 \le j \le \ell} \alpha_j \cdot n^{a_j} \cdot b_j^n$ with $\ell, a_j \in \mathbb{N}$, $b_j \in \mathbb{A}$,[1] $\alpha_j \in \mathbb{A}[\mathcal{V}]$, and for all $\sigma : \mathcal{V} \cup \{n\} \to \mathbb{Z}$ with $\sigma(n) \ge n_0$ we have $\sigma(\texttt{cl}^{x_i}) = \sigma(\eta^n(x_i))$. Similarly, we call $\texttt{cl} = (\texttt{cl}^{x_1}, \ldots, \texttt{cl}^{x_d})$ a* closed form *of the update η (resp. for the loop (φ, η)) with start value n_0 if for all $1 \le i \le d$, $\texttt{cl}^{x_i}$ are closed forms for x_i with start value n_0.*

Example 2. In Sect. 3 we will show that for the loop (1), a closed form for x_1 (with start value 0) is $\texttt{cl}^{x_1} = \frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n$ where $\alpha = (1+3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2$. Here, $\overline{\alpha}$ denotes the complex conjugate of α, i.e., the sign of those monomials is flipped where the coefficient is a multiple of the imaginary unit i. A closed form for x_4 (also with start value 0) is $\texttt{cl}^{x_4} = x_4 + n \cdot (\frac{1}{6} + x_3 + x_3^2 - x_3 \cdot n - \frac{n}{2} + \frac{n^2}{3})$.

Our aim is to compute *bounds* on the sizes of variables and on the runtime. As in [8,19], we only consider bounds which are weakly monotonically increasing in all occurring variables. Their advantage is that we can compose them easily (i.e., if f and g increase monotonically, then so does $f \circ g$).

Definition 3 (Bounds). *The set of* bounds *$\mathcal{B}$ is the smallest set with $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\} \subseteq \mathcal{B}$, $\mathcal{V} \subseteq \mathcal{B}$, and $\{b_1 + b_2,\, b_1 \cdot b_2,\, k^{b_1}\} \subseteq \mathcal{B}$ for all $k \in \mathbb{N}$ and $b_1, b_2 \in \mathcal{B}$.*

Size bounds should be bounds on the values of variables up to the point where the loop guard is not satisfied anymore for the first time. To define size bounds, we introduce the *runtime complexity* of a loop (whereas we considered the runtime complexity of arbitrary integer programs in [8,19,27]). Let Σ denote the set of all states $\sigma : \mathcal{V} \to \mathbb{Z}$ and let $|\sigma|$ be the state with $|\sigma|(x) = |\sigma(x)|$ for all $x \in \mathcal{V}$.

Definition 4 (Runtime Complexity for Loops). *The* runtime complexity *of a loop (φ, η) is $\mathrm{rc} : \Sigma \to \overline{\mathbb{N}}$ with $\mathrm{rc}(\sigma) = \inf\{n \in \mathbb{N} \mid \sigma(\eta^n(\neg\varphi))\}$, where $\inf \varnothing = \omega$. An expression $r \in \mathcal{B}$ is a* runtime bound *if $|\sigma|(r) \ge \mathrm{rc}(\sigma)$ for all $\sigma \in \Sigma$.*

Example 5. The runtime complexity of the loop (1) is $\mathrm{rc}(\sigma) = \max(0, \sigma(x_3))$. For example, x_3 is a runtime bound, as $|\sigma|(x_3) \ge \max(0, \sigma(x_3))$ for all states $\sigma \in \Sigma$.

A *size bound* on a variable x is a bound on the absolute value of x after n iterations of the update η, where n is bounded by the runtime complexity. In contrast to the definition of size bounds for transitions in integer programs from [8], Definition 6 requires that size bounds also hold *before* evaluating the loop.

[1] $\mathbb{A}$ is the set of algebraic numbers, i.e., the field of all roots of polynomials in $\mathbb{Z}[x]$.

Definition 6 (Size Bounds for Loops). *$\mathcal{SB} : \mathcal{V} \to \mathcal{B}$ is a* size bound *for (φ, η) if for all $x \in \mathcal{V}$ and all $\sigma \in \Sigma$, we have $|\sigma|(\mathcal{SB}(x)) \geq \sup\{|\sigma(\eta^n(x))| \mid n \leq \mathrm{rc}(\sigma)\}$.*

For any algebraic number $c \in \mathbb{A}$, as usual $\lceil |c| \rceil$ is the smallest natural number which is greater or equal to c's absolute value. Similarly, for any poly-exponential expression $p = \sum_j (\sum_i c_{i,j} \cdot \beta_{i,j}) \cdot n^{a_j} \cdot b_j^n$ where $c_{i,j} \in \mathbb{A}$ and the $\beta_{i,j}$ are normalized monomials of the form $x_1^{e_1} \cdot \ldots \cdot x_d^{e_d}$, $\lceil |p| \rceil$ denotes $\sum_j (\sum_i \lceil |c_{i,j}| \rceil \cdot \beta_{i,j}) \cdot n^{a_j} \cdot \lceil |b_j| \rceil^n$.

We now determine size bounds by over-approximating the closed form $\mathtt{cl}^x$ by the non-negative, weakly monotonically increasing function $\lceil |\mathtt{cl}^x| \rceil$. Then we substitute n by a runtime bound r (denoted by "$[n/r]$"). Due to the monotonicity, this results in a bound on the size of x not only at the end of the loop, but also during the iterations of the loop. Since the closed form is only valid for n iterations with $n \geq n_0$, we ensure that our size bound is also correct for less than n_0 iterations by symbolically evaluating the update, where we over-approximate maxima by sums. As mentioned, see [28] for the proofs of all new results.

Theorem 7 (Size Bounds for Loops with Closed Forms). *Let $\mathtt{cl}$ be a closed form for the loop (φ, η) with start value n_0 and let $r \in \mathcal{B}$ be a runtime bound. Then the (absolute) size of $x \in \mathcal{V}$ is bounded by $\mathtt{sb}^x = \lceil |\mathtt{cl}^x| \rceil [n/r] + \sum_{0 \leq i < n_0} |\eta^i(x)|$. Hence, the function $\mathcal{SB}$ with $\mathcal{SB}(x) = \mathtt{sb}^x$ for all $x \in \mathcal{V}$ is a size bound for (φ, η).*

Example 8. As mentioned, for the loop (1), a closed form for x_1 with start value 0 is $\mathtt{cl}^{x_1} = \frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n$ where $\alpha = (1 + 3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2$. Hence, $\lceil |\mathtt{cl}^{x_1}| \rceil = \lceil |\frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n| \rceil = (\lceil |\frac{1+3\mathrm{i}}{2}| \rceil \cdot x_1 + \lceil |\mathrm{i}| \rceil \cdot x_2) \cdot \lceil | - \mathrm{i}| \rceil^n + (\lceil |\frac{1-3\mathrm{i}}{2}| \rceil \cdot x_1 + \lceil | - \mathrm{i}| \rceil \cdot x_2) \cdot \lceil |\mathrm{i}| \rceil^n = 4 \cdot x_1 + 2 \cdot x_2$, as $\lceil |\frac{1+3\mathrm{i}}{2}| \rceil = \lceil |\frac{1-3\mathrm{i}}{2}| \rceil = \left\lceil \frac{\sqrt{10}}{2} \right\rceil = 2$ and $\lceil |\mathrm{i}| \rceil = \lceil | - \mathrm{i}| \rceil = 1$. So our approach infers *linear* size bounds for x_1 and x_2 (the similar computations for x_2 are omitted) while [8] only infers exponential size bounds.

As this over-approximation does not depend on n, it directly yields a size bound, i.e., $\mathtt{sb}^{x_1} = \lceil |\mathtt{cl}^{x_1}| \rceil$. In contrast, in the over-approximation $\lceil |\mathtt{cl}^{x_4}| \rceil = x_4 + n \left(1 + x_3 + x_3^2 + x_3 \cdot n + n + n^2\right)$, we have to replace n by a runtime bound like x_3. Thus, we obtain the overall size bound $\mathtt{sb}^{x_4} = x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3$.

Although this section focused on closed forms which are poly-exponential expressions, our technique is applicable to all loops where we can compute over-approximating bounds for the closed form and the runtime complexity. For example, the update $\eta(x) = x^2$ has the closed form $x^{(2^n)}$, but it does not admit a poly-exponential closed form due to x's super-exponential growth. However, by instantiating n by a runtime bound, we can still compute a size bound for this update. The reason for focusing on poly-exponential expressions is that we can compute such a closed form for all so-called *solvable loops* automatically, see Sect. 3.

3 Size and Runtime Bounds for Solvable Loops

In this section, we present a class of loops where our technique of Theorem 7 is "complete". The technique relies on the computation of suitable closed forms and of runtime bounds. In Sect. 3.1, we show that poly-exponential closed forms can be computed for all *solvable loops* [17,23,25,26,32,36]. Then we prove in Sect. 3.2 that finite runtime bounds are computable for all terminating solvable loops with only periodic rational eigenvalues.

A loop (φ, η) is *solvable* if η is a *solvable update* (see Definition 9 below for a formal definition), which partitions $\mathcal{V}$ into blocks $\mathcal{S}_1, \ldots, \mathcal{S}_m$ (and loop guards φ are not relevant for closed forms). Each block allows updates with *cyclic dependencies* between its variables and *non-linear* dependencies on variables in blocks with lower indices.

Definition 9 (Solvable Update [17,23,25,26,32,36]**).** *An update* $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ *is* solvable *if there exists a partition* $\mathcal{S}_1, \ldots, \mathcal{S}_m$ *of* $\{x_1, \ldots, x_d\}$ *such that for all* $1 \leq i \leq m$ *we have* $\boldsymbol{\eta}_{\mathcal{S}_i} = A_{\mathcal{S}_i} \cdot \boldsymbol{x}_{\mathcal{S}_i} + \boldsymbol{p}_{\mathcal{S}_i}$ *for an* $A_{\mathcal{S}_i} \in \mathbb{Z}^{|\mathcal{S}_i| \times |\mathcal{S}_i|}$ *and a* $\boldsymbol{p}_{\mathcal{S}_i} \in \mathbb{Z}[\bigcup_{j<i} \mathcal{S}_j]^{|\mathcal{S}_i|}$*, where* $\boldsymbol{\eta}_{\mathcal{S}_i}$ *is the vector of all* $\eta(x_j)$ *and* $\boldsymbol{x}_{\mathcal{S}_i}$ *is the vector of all* x_j *with* $j \in \mathcal{S}_i$*. The eigenvalues of a solvable loop are defined as the union of the eigenvalues of all matrices* $A_{\mathcal{S}_i}$*. The loop is* homogeneous *if* $\boldsymbol{p}_{\mathcal{S}_i} = \mathbf{0}$ *for all* $1 \leq i \leq m$*.*

Example 10. The loop (1) is an example for a solvable loop using the partition $\mathcal{S}_1 = \{x_1, x_2\}$, $\mathcal{S}_2 = \{x_3\}$, and $\mathcal{S}_3 = \{x_4\}$.

The crucial idea for our results in Sect. 3.1 and 3.2 is to reduce the problem of finding closed forms and runtime bounds from solvable loops to *triangular weakly non-linear* loops (*twn-loops*) [16,17,20]. A *twn-update* is a solvable update where each block $\mathcal{S}_j$ has cardinality one. Thus, a twn-update is *triangular*, i.e., the update of a variable does not depend on variables with higher indices. Furthermore, the update is *weakly non-linear*, i.e., a variable does not occur non-linear in its own update. We are mainly interested in loops over $\mathbb{Z}$, but to handle solvable updates, we will transform them into twn-updates with coefficients from $\mathbb{A}$.

Definition 11 (TWN-Update [16,17,20]**).** *An update* $\eta : \mathcal{V} \to \mathbb{A}[\mathcal{V}]$ *is* twn *if for all* $1 \leq i \leq d$ *we have* $\eta(x_i) = c_i \cdot x_i + p_i$ *for some* $c_i \in \mathbb{A}$ *and some polynomial* $p_i \in \mathbb{A}[x_1, \ldots, x_{i-1}]$*. A loop with a twn-update is called a* twn-loop*.*

Clearly, (1) is not a twn-loop due to the cyclic dependency between x_1 and x_2.

3.1 Closed Forms for Solvable Loops

Lemma 12 (which extends [17, Thm. 16] from solvable updates with real eigenvalues to arbitrary solvable updates) illustrates that one can transform any <u>s</u>olvable update η_s into a <u>t</u>wn-update η_t by an automorphism ϑ. Here, ϑ is induced by the change-of-basis matrix of the Jordan normal form of each block of η_s. Note that the Jordan normal form is always computable in polynomial time (see [9]).

Lemma 12 (Transforming Solvable Updates (see [17], Thm. 16). *Let η_s be a solvable update. Then $\vartheta : \mathcal{V} \to \mathbb{A}[\mathcal{V}]$ is an automorphism, where ϑ is defined by $\vartheta(\mathcal{S}) = P \cdot \boldsymbol{x}_{\mathcal{S}}$ for each block $\mathcal{S}$, where $J(A_{\mathcal{S}}) = P \cdot A_{\mathcal{S}} \cdot P^{-1}$ is the Jordan normal form of $A_{\mathcal{S}}$. Furthermore, $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ is a twn-update.*

Example 13 To illustrate Lemma 12, we transform the solvable update η_s of (1) into a twn-update η_t. As the blocks $\mathcal{S}_2 = \{x_3\}$ and $\mathcal{S}_3 = \{x_4\}$ have cardinality one, we only have to consider $\mathcal{S}_1 = \{x_1, x_2\}$. The restriction of η_s to $\mathcal{S}_1$ is $\binom{x_1}{x_2} \leftarrow A_{\mathcal{S}_1} \cdot \binom{x_1}{x_2}$ with $A_{\mathcal{S}_1} = \left(\begin{smallmatrix} 3 & 2 \\ -5 & -3 \end{smallmatrix}\right)$. So we get the Jordan normal form $J(A_{\mathcal{S}_1}) = P \cdot A_{\mathcal{S}_1} \cdot P^{-1} = \left(\begin{smallmatrix} -\mathrm{i} & 0 \\ 0 & \mathrm{i} \end{smallmatrix}\right)$ where $P = \left(\begin{smallmatrix} -\frac{5}{2}\mathrm{i} & \frac{1}{2}(1-3\mathrm{i}) \\ \frac{5}{2}\mathrm{i} & \frac{1}{2}(1+3\mathrm{i}) \end{smallmatrix}\right)$ and $P^{-1} = \left(\begin{smallmatrix} \frac{1}{5}(\mathrm{i}-3) & -\frac{1}{5}(\mathrm{i}+3) \\ 1 & 1 \end{smallmatrix}\right)$. Thus, we have the following automorphism ϑ and its inverse ϑ^{-1}:

$$\vartheta \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = P \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -\frac{5}{2}\mathrm{i} \cdot x_1 + \frac{1}{2}(1-3\mathrm{i}) \cdot x_2 \\ \frac{5}{2}\mathrm{i} \cdot x_1 + \frac{1}{2}(1+3\mathrm{i}) \cdot x_2 \end{pmatrix}, \qquad \vartheta \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}$$

$$\vartheta^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = P^{-1} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{5}(\mathrm{i}-3) \cdot x_1 - \frac{1}{5}(\mathrm{i}+3) \cdot x_2 \\ x_1 + x_2 \end{pmatrix}, \qquad \vartheta^{-1} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}$$

Hence, $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ is the following twn-update:

$$\eta_t(x_1) = -\mathrm{i} \cdot x_1, \quad \eta_t(x_2) = \mathrm{i} \cdot x_2, \quad \eta_t(x_3) = x_3 - 1, \quad \eta_t(x_4) = x_4 + x_3^2$$

The reason for transforming solvable updates to twn-updates is that for the latter, we can re-use our previous algorithm from [16] to compute poly-exponential closed forms. While [16] only considered updates with linear arithmetic over $\mathbb{Z}$, it can directly be extended to twn-updates over $\mathbb{A}$.

Lemma 14 (Closed Forms for TWN-Updates (see [16])). *Let η be a twn-update. Then a (poly-exponential) closed form is computable for η.*

Example 15. For η_t from Example 13, we obtain the following closed form (with start value 0): $\mathtt{cl}_t = ((-\mathrm{i})^n \cdot x_1, \mathrm{i}^n \cdot x_2, x_3 - n, x_4 + n(\frac{1}{6} + x_3 + x_3^2 - x_3 \cdot n - \frac{n}{2} + \frac{n^2}{3}))$.

So to obtain a closed form of a solvable update η_s, we first transform it into a twn-update η_t via Lemma 12, and then compute the closed form $\mathtt{cl}_t$ of η_t (Lemma 14). We now show how to obtain a closed form for η_s from $\mathtt{cl}_t$.

Theorem 16 (Closed Forms for Solvable Updates). *Let η_s be a solvable update and ϑ be an automorphism as in Lemma 12 such that $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ is a twn-update. If $\mathtt{cl}_t$ is a closed form of η_t with start value n_0, then $\mathtt{cl}_s = \vartheta \circ \mathtt{cl}_t \circ \vartheta^{-1}$ is a closed form of η_s with start value n_0.*

Example 17. In Example 13 we transformed η_s into the twn-update η_t via an automorphism ϑ and in Example 15, we gave a closed form $\mathtt{cl}_t$ of η_t. Thus, by Theorem 16, we can infer a closed form $\mathtt{cl}_s = \vartheta \circ \mathtt{cl}_t \circ \vartheta^{-1}$ of η_s. For example, we compute a closed form for x_1 with start value 0 ($\mathtt{cl}_s^{x_2}$ can be inferred in a similar way):

$$\begin{aligned} \mathtt{cl}_s^{x_1} &= \left(\tfrac{1}{5}(\mathrm{i}-3) \cdot x_1 - \tfrac{1}{5}(\mathrm{i}+3) \cdot x_2\right) \; [v/\mathtt{cl}_t^v \mid v \in \mathcal{V}] \; [v/\vartheta(v) \mid v \in \mathcal{V}] \\ &= \left(\tfrac{1}{5}(\mathrm{i}-3) \cdot (-\mathrm{i})^n \cdot x_1 - \tfrac{1}{5}(\mathrm{i}+3) \cdot \mathrm{i}^n \cdot x_2\right) \; [v/\vartheta(v) \mid v \in \mathcal{V}] \\ &= \tfrac{1}{2}(\underbrace{(1+3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2}_{\alpha}) \cdot (-\mathrm{i})^n + \tfrac{1}{2}(\underbrace{(1-3\mathrm{i}) \cdot x_1 - 2\mathrm{i} \cdot x_2}_{\overline{\alpha}}) \cdot \mathrm{i}^n. \end{aligned}$$

3.2 Periodic Rational Solvable Loops

In Sect. 3.1, we discussed how to compute closed forms for solvable updates (by transforming them to twn-updates). However to compute size bounds, we have to instantiate the variable n in the closed forms by runtime bounds (Theorem 7). In [20], it was shown that (polynomial) runtime bounds can always be computed for terminating twn-loops over the integers. However, in general, transforming solvable loops via Lemma 12 yields twn-updates which may contain algebraic (complex) numbers. We now show that for the subclass of terminating *periodic rational* solvable loops, our approach is "complete" (i.e., finite runtime bounds and thus, also finite size bounds are always computable).

Definition 18 (Periodic Rational [25]). *A number $\lambda \in \mathbb{A}$ is* periodic rational *if $\lambda^p \in \mathbb{Q}$ for some $p \in \mathbb{N}$ with $p > 0$. The* period *of λ is the smallest such p with $\lambda^p \in \mathbb{Q}$. A solvable loop is* periodic rational *(i.e., it is a* prs loop*) with period p if all its eigenvalues λ are periodic rational and p is the least common multiple of all their periods. A prs loop is a* unit *prs loop if $|\lambda| \leq 1$ for all its eigenvalues λ.*

So i, $-$i, and $\sqrt{2}\cdot$i are periodic rational with period 2, while $\sqrt{2}+$i is not periodic rational. The following lemma from [25] gives a bound on the period of prs loops and thus yields an algorithm to detect prs loops and to compute their period.

Lemma 19 (Bound on the Period [25]). *Let $A \in \mathbb{Z}^{n\times n}$. If λ is a periodic rational eigenvalue of A with period p, then $p \leq n^3$.*

Now we show that by *chaining* (i.e., by performing p iterations of a prs loop with period p in a single step), one can transform any prs loop into a solvable loop with only integer eigenvalues. Then, our previous results on twn-loops [17,20] can be used to infer runtime bounds for these loops.

Definition 20 (Chaining Loops). *Let $L = (\varphi, \eta)$ be a loop and $p \in \mathbb{N} \setminus \{0\}$. Then $L_p = (\varphi_p, \eta_p)$ results from iterating L p times, i.e., $\varphi_p = \varphi \wedge \eta(\varphi) \wedge \eta(\eta(\varphi)) \wedge \ldots \wedge \eta^{p-1}(\varphi)$ and $\eta_p(v) = \eta^p(v)$ for all $v \in \mathcal{V}$.*

Example 21. The eigenvalues $\pm$i of (1) have period 2. Chaining yields $(\varphi \wedge \eta(\varphi), \eta^2)$:

$$\textbf{while } (x_3 > 0 \wedge x_3 > 1) \textbf{ do } (x_1, x_2, x_3, x_4) \leftarrow (-x_1, -x_2, x_3 - 2, x_4 + (x_3 - 1)^2 + x_3^2) \quad (2)$$

Due to Lemma 12 we can transform every solvable update into a twn-update by a (linear) automorphism ϑ. For prs loops, ϑ's range can be restricted to $\mathbb{Q}[\mathcal{V}]$, i.e., one does not need algebraic numbers. So, we first chain the prs loop L and then compute a $\mathbb{Q}$-automorphism ϑ transforming the chained loop L_p into a twn-loop L_t via Lemma 12. Then we can infer a runtime bound for L_t as in [20]. The reason is that all factors c_i in the update of L_t are integers and thus, we can compute a closed form $\sum_j \alpha_j \cdot n^{a_j} \cdot b_j^n$ such that $\alpha_j \in \mathbb{Q}[\mathcal{V}]$ and $b_j \in \mathbb{Z}$. Afterwards, the runtime bound for L_t can be lifted to a runtime bound for the original loop

by reconsidering the automorphism ϑ. Similarly, in order to prove termination of the prs loop L, we analyze termination of L_t on $\vartheta(\mathbb{Z}^d) = \{\vartheta(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathbb{Z}^d\}$.[2]

Lemma 22 (Runtime Bounds for PRS Loops). *Let L be a prs loop with period p and let $L_p = (\varphi_p, \eta_p)$ result from chaining as in Definition 20. From η_p, one can compute a linear automorphism $\vartheta : \mathcal{V} \to \mathbb{Q}[\mathcal{V}]$ as in Lemma 12, such that:*

(a) L_p is solvable and only has integer eigenvalues.
(b) $(\vartheta^{-1} \circ \eta_p \circ \vartheta) : \mathcal{V} \to \mathbb{Q}[\mathcal{V}]$ is a twn-update as in Definition 11 such that all $c_i \in \mathbb{Z}$.
(c) $L_t = (\varphi_t, \eta_t)$ with $\varphi_t = \vartheta^{-1}(\varphi_p)$ and $\eta_t = \vartheta^{-1} \circ \eta_p \circ \vartheta$ is a twn-loop. Moreover, the following holds:
- *L terminates on $\mathbb{Z}^d$ iff*
- *L_p terminates on $\mathbb{Z}^d$ iff*
- *L_t terminates on $\vartheta(\mathbb{Z}^d) = \{\vartheta(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathbb{Z}^d\}$.*

(d) If r is a runtime bound[3] for L_t, then $p \cdot \lceil |\vartheta(r)| \rceil + p - 1$ is a runtime bound for L.

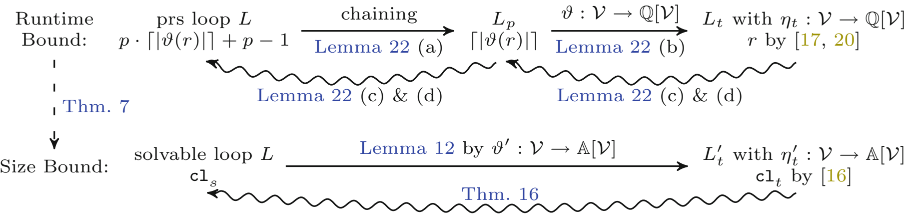


Fig. 1. Illustration of Runtime and Size Bound Computations

Since we can detect prs loops and their periods by Lemma 19, Lemma 22 allows us to compute runtime bounds for all terminating prs loops. This is illustrated in Fig. 1: For runtime bounds, L is transformed to L_p by chaining and L_p is transformed further to L_t by an automorphism ϑ. The runtime bound r for L_t can then be transformed into a runtime bound for L_p and further into a runtime bound for L. For size bounds, L is directly transformed to a twn-loop L'_t by an automorphism ϑ'. The closed form $\texttt{cl}_t$ obtained for L'_t is transformed via the automorphism ϑ' into a closed form $\texttt{cl}_s$ for L. Then the runtime bound for L is inserted into this closed form to yield a size bound for L. So in Fig. 1, standard arrows denote transformations of loops and wavy arrows denote transformations of runtime bounds or closed forms.

[2] By [17], termination of L_t on $\vartheta(\mathbb{Z}^d)$ is reducible to invalidity of a formula $\exists \boldsymbol{x} \in \mathbb{Q}^d . \psi_{\vartheta(\mathbb{Z}^d)} \land \xi_{L_t}$. Here, $\psi_{\vartheta(\mathbb{Z}^d)}$ holds iff $\boldsymbol{x} \in \vartheta(\mathbb{Z}^d)$ and ξ_{L_t} holds iff L_t does not terminate on $\boldsymbol{x}$. As shown in [17], non-termination of linear twn-loops with integer eigenvalues is NP-complete and it is semi-decidable for twn-loops with non-linear arithmetic.

[3] More precisely, $|\sigma|(r) \geq \inf\{n \in \mathbb{N} \mid \sigma(\eta_t^n(\neg\varphi_t))\}$ must hold for all $\sigma : \mathcal{V} \to \vartheta(\mathbb{Z}^d)$.

Theorem 23 (Completeness of Size and Runtime Bound Computation for Terminating PRS Loops). *For all terminating prs loops, polynomial runtime bounds and finite size bounds are computable. For terminating unit prs loops, all these size bounds are polynomial as well.*

Example 24. For the loop L from (1), we computed L_p for $p = 2$ in (2), see Example 21. As L_p is already a twn-loop, we can use the technique of [20] (implemented in our tool KoAT) to obtain the runtime bound x_3 for L_p. Lemma 22 yields the runtime bound $2 \cdot x_3 + 1$ for the original loop (1). Of course, here one could also use (incomplete) approaches based on linear ranking functions (also implemented in KoAT, see, e.g., [8,19]) to directly infer the tighter runtime bound x_3 for the loop (1).

4 Size Bounds for Integer Programs

Up to now, we focused on *isolated* loops. In the following, we incorporate our complete approach from Sect. 2 and 3 into the setting of general *integer programs* where most questions regarding termination or complexity are undecidable. Formally, an integer program is a tuple $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ with a finite set of variables $\mathcal{V}$, a finite set of locations $\mathcal{L}$, a fixed initial location $\ell_0 \in \mathcal{L}$, and a finite set of transitions $\mathcal{T}$. A *transition* is a 4-tuple $(\ell, \varphi, \eta, \ell')$ with a *start location* $\ell \in \mathcal{L}$, *target location* $\ell' \in \mathcal{L} \setminus \{\ell_0\}$, *guard* $\varphi \in \mathcal{F}(\mathcal{V})$, and *update* $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$. To simplify the presentation, we do not consider "temporary" variables (whose update is non-deterministic), but the approach can easily be extended accordingly. Transitions $(\ell_0, _, _, _)$ are called *initial* and $\mathcal{T}_0$ denotes the set of all initial transitions.

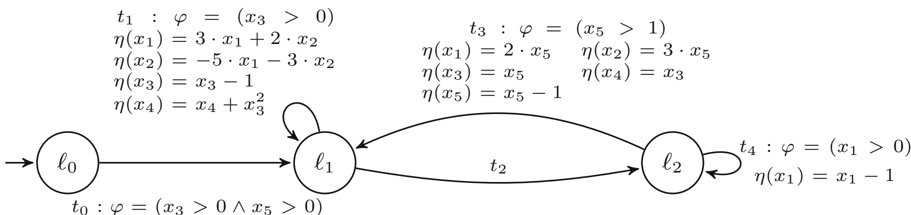


Fig. 2. An Integer Program with Non-Linear Size Bounds

Example 25. In the integer program of Fig. 2, we omitted identity updates $\eta(v) = v$ and guards where φ is `true`. Here, $\mathcal{V} = \{x_1, \ldots, x_5\}$ and $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, where ℓ_0 is the initial location. Note that the loop in (1) *corresponds* to transition t_1.

Definition 26 (Correspondence between Loops and Transitions). *Let $t = (\ell, \varphi, \eta, \ell)$ be a transition with $\varphi \in \mathcal{F}(\mathcal{V}')$ for some variables $\mathcal{V}' \subseteq \mathcal{V}$ such that $\eta(x) = x$ for all $x \in \mathcal{V} \setminus \mathcal{V}'$ and $\eta(x) \in \mathbb{Z}[\mathcal{V}']$ for all $x \in \mathcal{V}'$. A loop (φ', η')*

with $\varphi' \in \mathcal{F}(\{x_1,\dots,x_d\})$ *and* $\eta' : \{x_1,\dots,x_d\} \to \mathbb{Z}[\{x_1,\dots,x_d\}]$ corresponds *to the transition* t *via the variable renaming* $\pi : \{x_1,\dots,x_d\} \to \mathcal{V}'$ *if* φ *is* $\pi(\varphi')$ *and for all* $1 \le i \le d$ *we have* $\eta(\pi(x_i)) = \pi(\eta'(x_i))$.

To define the semantics of integer programs, an evaluation step moves from one configuration $(\ell,\sigma) \in \mathcal{L} \times \Sigma$ to another configuration (ℓ',σ') via a transition $(\ell,\varphi,\eta,\ell')$ where $\sigma(\varphi)$ holds. Here, σ' is obtained by applying the update η on σ. From now on, we fix an integer program $\mathcal{P} = (\mathcal{V},\mathcal{L},\ell_0,\mathcal{T})$.

Definition 27 (Evaluation of Programs). *For configurations* (ℓ,σ), (ℓ',σ') *and* $t = (\ell_t,\varphi,\eta,\ell'_t) \in \mathcal{T}$, $(\ell,\sigma) \to_t (\ell',\sigma')$ *is an* evaluation *step if* $\ell = \ell_t$, $\ell' = \ell'_t$, $\sigma(\varphi) = \texttt{true}$, *and* $\sigma(\eta(v)) = \sigma'(v)$ *for all* $v \in \mathcal{V}$. *Let* $\to_{\mathcal{T}} = \bigcup_{t\in\mathcal{T}} \to_t$, *where we also write* $\to$ *instead of* $\to_t$ *or* $\to_{\mathcal{T}}$. *Let* $(\ell_0,\sigma_0) \to^k (\ell_k,\sigma_k)$ *abbreviate* $(\ell_0,\sigma_0) \to \ldots \to (\ell_k,\sigma_k)$ *and let* $(\ell,\sigma) \to^* (\ell',\sigma')$ *if* $(\ell,\sigma) \to^k (\ell',\sigma')$ *for some* $k \ge 0$.

Example 28. If we encode states as tuples $(\sigma(x_1),\dots,\sigma(x_5)) \in \mathbb{Z}^5$, then $(-6,-8,2,1,1) \to_{t_0} (-6,-8,2,1,1) \to^2_{t_1} (6,8,0,6,1) \to_{t_2} (6,8,0,6,1) \to^6_{t_4} (0,8,0,6,1)$.

Now we define size bounds for variables v after evaluating a transition t: $\mathcal{SB}(t,v)$ is a *size bound* for v w.r.t. t if for any run starting in $\sigma_0 \in \Sigma$, $|\sigma_0|(\mathcal{SB}(t,v))$ is greater or equal to the largest absolute value of v after evaluating t.

Definition 29 (Size Bounds [8,19]). *A function* $\mathcal{SB} : (\mathcal{T} \times \mathcal{V}) \to \mathcal{B}$ *is a* (global) size bound *for the program* $\mathcal{P}$ *if for all* $(t,x) \in \mathcal{T}\times\mathcal{V}$ *and all states* $\sigma_0 \in \Sigma$ *we have* $|\sigma_0|(\mathcal{SB}(t,x)) \ge \sup\{|\sigma'(x)| \mid \exists\, \ell' \in \mathcal{L}.\ (\ell_0,\sigma_0)\ (\to^* \circ \to_t)\ (\ell',\sigma')\}$.

Later in Lemma 35, we will compare the notion of size bounds for transitions in a program from Definition 29 to our earlier notion of size bounds for loops from Definition 6.

Example 30. As an example, we give size bounds for the transitions t_0 and t_3 in Fig. 2. Since t_0 does not change any variables, a size bound is $\mathcal{SB}(t_0,x_i) = x_i$ for all $1 \le i \le 5$. Note that the value of x_5 is never increased and is bounded from below by 0 in any run through the program. Thus, $\mathcal{SB}(t_3,x_3) = x_5 = \mathcal{SB}(t_3,x_5)$. Similarly, we have $\mathcal{SB}(t_3,x_1) = 2\cdot x_5$, $\mathcal{SB}(t_3,x_2) = 3\cdot x_5$, and $\mathcal{SB}(t_3,x_4) = x_3$.

To infer size bounds for transitions as in Definition 29 automatically, we lift *local* size bounds (i.e., size bounds which only hold for a subprogram with transitions $\mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$) to global size bounds for the *complete* program. For the subprogram, one considers runs which start after evaluating an *entry transition* of $\mathcal{T}'$.

Definition 31 (Entry Transitions [8]). *Let* $\varnothing \ne \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$. *The* entry transitions *of* $\mathcal{T}'$ *are* $\mathcal{E}_{\mathcal{T}'} = \{t \mid t = (_,_,_,\ell) \in \mathcal{T} \setminus \mathcal{T}'$ *and there is a* $(\ell,_,_,_) \in \mathcal{T}'\}$.

Example 32. For the program in Fig. 2, we have $\mathcal{E}_{\{t_1\}} = \{t_0, t_3\}$ and $\mathcal{E}_{\{t_4\}} = \{t_2\}$.

Definition 33 (Local Size Bounds). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$ and $t' \in \mathcal{T}'$. $\mathcal{SB}_{t'} : \mathcal{V} \to \mathcal{B}$ is a* local size bound *for t' w.r.t. $\mathcal{T}'$ if for all $x \in \mathcal{V}$ and all $\sigma \in \Sigma$:*[4] $|\sigma|(\mathcal{SB}_{t'}(x)) \geq \sup\{|\sigma'(x)| \mid \exists \ell' \in \mathcal{L}, (_,_,_,\ell) \in \mathcal{E}_{\mathcal{T}'}.\ (\ell,\sigma)\ (\to^*_{\mathcal{T}'} \circ \to_{t'})\ (\ell',\sigma')\}$.

Theorem 34 below yields a novel *modular* procedure to infer (global) size bounds from previously computed local size bounds. A local size bound for a transition t' w.r.t. a subprogram $\mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$ is lifted by inserting size bounds for all entry transitions. Again, this is possible because we only use weakly monotonically increasing functions as bounds. Here, "$b\,[v/p_v \mid v \in \mathcal{V}]$" denotes the bound which results from replacing every variable v by p_v in the bound b.

Theorem 34 (Lifting Local Size Bounds). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$, let $\mathcal{SB}_{t'}$ be a local size bound for a transition t' w.r.t. $\mathcal{T}'$ and let $\mathcal{SB} : (\mathcal{T} \times \mathcal{V}) \to \mathcal{B}$ be a size bound for $\mathcal{P}$. Let $\mathcal{SB}'(t',x) = \sum_{r \in \mathcal{E}_{\mathcal{T}'}} \mathcal{SB}_{t'}(x)\,[v/\mathcal{SB}(r,v) \mid v \in \mathcal{V}]$ and $\mathcal{SB}'(t,x) = \mathcal{SB}(t,x)$ for all $t' \neq t$. Then $\mathcal{SB}'$ is also a size bound for $\mathcal{P}$.*

To obtain local size bounds which can then be lifted via Theorem 34, we look for transitions t_L that correspond to a loop L and then we compute a size bound for L as in Sect. 2 and 3. The following lemma shows that size bounds for loops as in Definition 6 indeed yield local size bounds for the corresponding transitions.[5]

Lemma 35 (Local Size Bounds via Loops). *Let $\mathcal{SB}_L$ be a size bound for a loop L (as in Definition 6) which corresponds to a transition t_L via a variable renaming π. Then $\pi \circ \mathcal{SB}_L \circ \pi^{-1}$ is a local size bound for t_L w.r.t. $\{t_L\}$ (as in Definition 33).*

Example 36. $\mathcal{SB}_L(x_4) = x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3$ is a size bound for x_4 in the loop (1), see Example 8. This loop corresponds to transition t_1 in the program of Fig. 2. Since $\mathcal{E}_{\{t_1\}} = \{t_0, t_3\}$ by Example 32, Theorem 34 yields the following (non-linear) size bound for x_4 in the full program of Fig. 2 (see Example 30 for $\mathcal{SB}(t_0, v)$ and $\mathcal{SB}(t_3, v)$):

$$\begin{aligned} \mathcal{SB}(t_1, x_4) &= \mathcal{SB}_L(x_4)\,[v/\mathcal{SB}(t_0,v) \mid v \in \mathcal{V}] + \mathcal{SB}_L(x_4)\,[v/\mathcal{SB}(t_3,v) \mid v \in \mathcal{V}] \\ &= (x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3) + (x_3 + 3 \cdot x_5^3 + 2 \cdot x_5^2 + x_5) \\ &= 2 \cdot x_3 + 2 \cdot x_3^2 + 3 \cdot x_3^3 + x_4 + x_5 + 2 \cdot x_5^2 + 3 \cdot x_5^3 \end{aligned}$$

Analogously, we infer the remaining size bounds $\mathcal{SB}(t_1, x_i)$, e.g., $\mathcal{SB}(t_1, x_1) = (4 \cdot x_1 + 2 \cdot x_2)\,[v/\mathcal{SB}(t_0,v) \mid v \in \mathcal{V}] + (4 \cdot x_1 + 2 \cdot x_2)\,[v/\mathcal{SB}(t_3,v) \mid v \in \mathcal{V}] = 4 \cdot x_1 + 2 \cdot x_2 + 14 \cdot x_5$.

---

[4] To simplify the formalism, in this definition, we consider every possible configuration (ℓ, σ) and not only configurations which are reachable from the initial location ℓ_0.

[5] Local or global size bounds for transitions only have to hold if the transition is indeed taken. In contrast, size bounds for loops also have to hold if there is no loop iteration. This will be needed in Theorem 38 to compute local size bounds for simple cycles.

Our approach alternates between improving size and runtime bounds for individual transitions. We start with $\mathcal{SB}(t_0, x) = |\eta(x)|$ for initial transitions $t_0 \in \mathcal{T}_0$ where η is t_0's update, and $\mathcal{SB}(t, _) = \omega$ for $t \in \mathcal{T} \setminus \mathcal{T}_0$. Here, similar to the notion $\lceil |p| \rceil$ in Sect. 2, for every polynomial $p = \sum_j c_j \cdot \beta_j$ with normalized monomials β_j, $|p|$ is the polynomial $\sum_j |c_j| \cdot \beta_j$. To improve the size bounds of transitions that correspond to (possibly non-linear) solvable loops, we can use closed forms (Theorem 7) and the lifting via Theorem 34. Otherwise, we use an existing incomplete technique [8] to improve size bounds (where [8] essentially only succeeds for updates without non-linear arithmetic). In this way, we can automatically compute polynomial size bounds for all remaining transitions and variables in the program of Fig. 2 (e.g., we obtain $\mathcal{SB}(t_2, x_1) = \mathcal{SB}(t_1, x_1) = 4 \cdot x_1 + 2 \cdot x_2 + 14 \cdot x_5$).

Both the technique from [8] and our approach from Theorem 7 rely on runtime bounds to compute size bounds. On the other hand, as shown in [8,19,27], size bounds for "previous" transitions are needed to infer (global) runtime bounds for transitions in a program. For that reason, the alternated computation resp. improvement of global size and runtime bounds for the transitions is repeated until all bounds are finite. We will illustrate this in more detail in Sect. 5.

In Definition 26 and Lemma 35 we considered transitions with the same start and target location that directly correspond to loops. To increase the applicability of our approach, as in [27] now we consider so-called *simple cycles*, where iterations through the cycle can only be done in a unique way. So the cycle must not have subcycles and there must not be any indeterminisms concerning the next transition to be taken. Formally, $\mathcal{C} = \{t_1, \ldots, t_n\} \subseteq \mathcal{T}$ is a simple cycle if there are pairwise different locations $\ell_1, \ldots, \ell_n$ such that $t_i = (\ell_i, _, _, \ell_{i+1})$ for $1 \leq i \leq n-1$ and $t_n = (\ell_n, _, _, \ell_1)$. To handle simple cycles, we *chain* transitions.[6]

Definition 37 (Chaining (see, e.g., [27])). *Let $t_1, \ldots, t_n \in \mathcal{T}$ where $t_i = (\ell_i, \varphi_i, \eta_i, \ell_{i+1})$ for all $1 \leq i \leq n-1$. Then the transition $t_1 \star \ldots \star t_n = (\ell_1, \varphi, \eta, \ell_{n+1})$ results from* chaining $t_1, \ldots, t_n$ *where*

$$\varphi = \varphi_1 \wedge \eta_1(\varphi_2) \wedge \eta_2(\eta_1(\varphi_3)) \wedge \ldots \wedge \eta_{n-1}(\ldots \eta_1(\varphi_n) \ldots)$$
$$\eta(v) = \eta_n(\ldots \eta_1(v) \ldots) \text{ for all } v \in \mathcal{V}, \text{ i.e., } \eta = \eta_n \circ \ldots \circ \eta_1.$$

Now we want to compute a *local* size bound for the transition t_n w.r.t. a simple cycle $\mathcal{C} = \{t_1, \ldots, t_n\}$ where a loop L corresponds to $t_1 \star \ldots \star t_n$ via π. Then a size bound $\mathcal{SB}_L$ for the loop L yields the size bound $\pi \circ \mathcal{SB}_L \circ \pi^{-1}$ for t_n regarding runs through $\mathcal{C}$ starting in t_1. However, to obtain a local size bound $\mathcal{SB}_{t_n}$ w.r.t. $\mathcal{C}$, we have to consider runs starting after any entry transition $(_, _, _, \ell_i) \in \mathcal{E}_{\mathcal{C}}$. Hence, we use $|\, \eta_n(\ldots \eta_i(\pi(\mathcal{SB}_L(\pi^{-1}(x)))) \ldots) \,|$ for any $(_, _, _, \ell_i) \in \mathcal{E}_{\mathcal{C}}$. In this way, we also capture evaluations starting in ℓ_i, i.e., without evaluating the complete cycle.

[6] The chaining of a loop L in Definition 20 corresponds to $p-1$ chaining steps of a transition t_L via Definition 37, i.e., to $t_L \star \ldots \star t_L$.

Theorem 38 (Local Size Bounds for Simple Cycles). *Let* $\mathcal{C} = \{t_1, \ldots, t_n\} \subseteq \mathcal{T}$ *be a simple cycle and let* $\mathcal{SB}_L$ *be a size bound for a loop* L *which corresponds to* $t_1 \star \ldots \star t_n$ *via a variable renaming* π*. Then a* local size bound $\mathcal{SB}_{t_n}$ *for* t_n *w.r.t.* $\mathcal{C}$ *is* $\mathcal{SB}_{t_n}(x) = \sum_{1 \le i \le n, (_,_,_,\ell_i) \in \mathcal{E}_\mathcal{C}} |\, \eta_n(\ldots \eta_i(\pi(\mathcal{SB}_L(\pi^{-1}(x)))) \ldots) \,|$.

Example 39. As an example, in the program of Fig. 2 we replace $t_1 = (\ell_1, x_3 > 0, \eta_1, \ell_1)$ by $t_{1a} = (\ell_1, \texttt{true}, \eta_{1a}, \ell_1')$ and $t_{1b} = (\ell_1', x_3 > 0, \eta_{1b}, \ell_1)$ with a new location ℓ_1', where $\eta_{1a}(v) = \eta_1(v)$ for $v \in \{x_1, x_2\}$, $\eta_{1b}(v) = \eta_1(v)$ for $v \in \{x_3, x_4\}$, and η_{1a} resp. η_{1b} are the identity on the remaining variables. Then $\{t_{1a}, t_{1b}\}$ forms a simple cycle and Theorem 38 allows us to compute local size bounds $\mathcal{SB}_{t_{1b}}$ and $\mathcal{SB}_{t_{1a}}$ w.r.t. $\{t_{1a}, t_{1b}\}$, because the chained transitions $t_{1a} \star t_{1b} = t_1$ and $t_{1b} \star t_{1a}$ both correspond to the loop (1). They can then be lifted to global size bounds as in Example 36 using size bounds for the entry transitions $\mathcal{E}_{\{t_{1a}, t_{1b}\}} = \{t_0, t_3\}$.

This shows how we choose t' and $\mathcal{T}'$ when lifting local size bounds to global ones with Theorem 34: For a transition t' we search for a simple cycle $\mathcal{T}'$ such that chaining the cycle results in a twn- or suitable solvable loop and the size bounds of $\mathcal{E}_{\mathcal{T}'}$ are finite. For all other transitions, we compute size bounds as in [8].

5 Completeness of Size and Runtime Analysis for Programs

For individual loops, we showed in Theorem 23 that polynomial runtime bounds and finite size bounds are computable for all terminating prs loops. In this section, we discuss completeness of the size bound technique from the previous section and of termination and runtime complexity analysis for general integer programs. We show that for a large class of programs consisting of consecutive prs loops, in case of termination we can always infer finite runtime and size bounds.

To this end, we briefly recapitulate how size bounds are used to compute runtime bounds for general integer programs, and show that our new technique to infer size bounds also results in better runtime bounds. We call $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ a *(global) runtime bound* if for every transition $t \in \mathcal{T}$ and state $\sigma_0 \in \Sigma$, $|\sigma_0|(\mathcal{RB}(t))$ over-approximates the number of evaluations of t in any run starting in (ℓ_0, σ_0).

Definition 40 (Runtime Bound [8,19]). *A function* $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ *is a* (global) runtime bound *if for all* $t \in \mathcal{T}$ *and all states* $\sigma_0 \in \Sigma$*, we have* $|\sigma_0|(\mathcal{RB}(t)) \ge \sup\{n \in \mathbb{N} \mid \exists\, (\ell', \sigma').\ (\ell_0, \sigma_0)\ (\to_\mathcal{T}^* \circ \to_t)^n\ (\ell', \sigma')\}$.

For our example in Fig. 2, a global runtime bound for t_0, t_2, and t_3 is $\mathcal{RB}(t_0) = 1$ and $\mathcal{RB}(t_2) = \mathcal{RB}(t_3) = x_5$, as x_5 is bounded from below by t_3's guard $x_5 > 1$ and the value of x_5 decreases by 1 in t_3, and no transition increases x_5.

To infer global runtime bounds automatically, similar as for size bounds, we first consider a smaller subprogram $\mathcal{T}' \subseteq \mathcal{T}$ and compute *local runtime bounds* for non-empty subsets $\mathcal{T}'_> \subseteq \mathcal{T}'$. A local runtime bound measures how often a transition $t \in \mathcal{T}'_>$ can occur in a run through $\mathcal{T}'$ that starts after an entry transition $r \in \mathcal{E}_{\mathcal{T}'}$. Thus, local runtime bounds do not consider how many $\mathcal{T}'$-runs take place in a global run and they do not consider the sizes of the variables before starting a $\mathcal{T}'$-run. We lift these local bounds to global runtime bounds for the complete program afterwards.

Definition 41 (Local Runtime Bound [27]). *Let* $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$. $\mathcal{RB}_{\mathcal{T}'_>} \in \mathcal{B}$ *is a* local runtime bound *for* $\mathcal{T}'_>$ *w.r.t.* $\mathcal{T}'$ *if for all* $t \in \mathcal{T}'_>$*, all* $r \in \mathcal{E}_{\mathcal{T}'}$ *with* $r = (\ell, _, _, _)$*, and all* $\sigma \in \Sigma$*, we have* $|\sigma|(\mathcal{RB}_{\mathcal{T}'_>}) \geq \sup\{n \in \mathbb{N} \mid \exists\, \sigma_0, (\ell', \sigma').\ (\ell_0, \sigma_0) \to^*_{\mathcal{T}} \circ \to_r (\ell, \sigma)\ (\to^*_{\mathcal{T}'} \circ \to_t)^n\ (\ell', \sigma')\}$.

Example 42. In Fig. 2, local runtime bounds for $\mathcal{T}'_> = \mathcal{T}' = \{t_1\}$ and for $\mathcal{T}'_> = \mathcal{T}' = \{t_4\}$ are $\mathcal{RB}_{\{t_1\}} = x_3$ and $\mathcal{RB}_{\{t_4\}} = x_1$. Local runtime bounds can often be inferred automatically by approaches based on ranking functions (see, e.g., [8]) or by the complete technique for terminating prs loops (see Theorem 23).

If we have a local runtime bound $\mathcal{RB}_{\mathcal{T}'_>}$ w.r.t. $\mathcal{T}'$, then setting $\mathcal{RB}(t)$ to $\sum_{r \in \mathcal{E}_{\mathcal{T}'}} \mathcal{RB}(r) \cdot (\mathcal{RB}_{\mathcal{T}'_>}\,[v/\mathcal{SB}(r,v) \mid v \in \mathcal{V}])$ for all $t \in \mathcal{T}'_>$ yields a global runtime bound [27]. Here, we over-approximate the number of local $\mathcal{T}'$-runs which are started by an entry transition $r \in \mathcal{E}_{\mathcal{T}'}$ by an already computed global runtime bound $\mathcal{RB}(r)$. Moreover, we instantiate each $v \in \mathcal{V}$ by a size bound $\mathcal{SB}(r, v)$ to consider the size of v before a local $\mathcal{T}'$-run is started. So as mentioned in Sect. 4, we need runtime bounds to infer size bounds (see Theorem 7 and the inference of global size bounds in [8]), and on the other hand we need size bounds to compute runtime bounds. Thus, our implementation alternates between size bound and runtime bound computations (see [8,27] for a more detailed description of this alternation).

Example 43. Based on the local runtime bounds in Example 42, we can compute the remaining global runtime bounds for our example. We obtain $\mathcal{RB}(t_1) = \mathcal{RB}(t_0) \cdot (x_3\,[v/\mathcal{SB}(t_0, v) \mid v \in \mathcal{V}]) + \mathcal{RB}(t_3) \cdot (x_3\,[v/\mathcal{SB}(t_3, v) \mid v \in \mathcal{V}]) = x_3 + x_5^2$ and $\mathcal{RB}(t_4) = \mathcal{RB}(t_2) \cdot (x_1\,[v/\mathcal{SB}(t_2, v) \mid v \in \mathcal{V}]) = x_5 \cdot (4 \cdot x_1 + 2 \cdot x_2 + 14 \cdot x_5)$. Thus, overall we have a quadratic runtime bound $\sum_{1 \leq i \leq 5} \mathcal{RB}(t_i)$. Note that it is due to our new size bound technique from Sect. 2–4 that we obtain polynomial runtime bounds in this example. In contrast, to the best of our knowledge, all other state-of-the-art tools fail to infer polynomial size or runtime bounds for this example. Similarly, if one modifies t_4 such that instead of x_1, x_4 is decreased as long as $x_4 > 0$ holds, then our approach again yields a polynomial runtime bound, whereas none of the other tools can infer finite runtime bounds.

Finally, we state our completeness results for integer programs. For a set $\mathcal{C} \subseteq \mathcal{T}$ and $\ell, \ell' \in \mathcal{L}$, let $\ell \rightsquigarrow_{\mathcal{C}} \ell'$ hold iff there is a transition $(\ell, _, _, \ell') \in \mathcal{C}$. We say that $\mathcal{C}$ is a *component* if we have $\ell \rightsquigarrow^+_{\mathcal{C}} \ell'$ for all locations ℓ, ℓ' occurring in $\mathcal{C}$, where $\rightsquigarrow^+_{\mathcal{C}}$ is the transitive closure of $\rightsquigarrow_{\mathcal{C}}$. So in particular, we must also have

$\ell \rightsquigarrow_{\mathcal{C}}^{+} \ell$ for all locations ℓ in the transitions of $\mathcal{C}$. We call an integer program *simple* if every component is a simple cycle that is "reachable" from any initial state.

Definition 44 (Simple Integer Program). *An integer program* $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ *is* simple *if every component* $\mathcal{C} \subseteq \mathcal{T}$ *is a simple cycle, and for every entry transition* $(_, _, _, \ell) \in \mathcal{E}_{\mathcal{C}}$ *and every* $\sigma_0 \in \Sigma$*, there is an evaluation* $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{*} (\ell, \sigma_0)$*.*

In Fig. 2, $\mathcal{T} \setminus \{t_0\}$ is a component that is no simple cycle. However, if we remove t_3 and replace t_0's guard by `true`, then the resulting program $\mathcal{P}'$ is simple (but not linear). A simple program terminates iff each of its isolated simple cycles terminates. Thus, if we can prove termination for every simple cycle, then the overall program terminates. Hence, if after chaining, every simple cycle corresponds to a linear, unit prs loop, then we can decide termination and infer polynomial runtime and size bounds for the overall integer program. For terminating, non-unit prs loops, runtime bounds are still polynomial but size bounds can be exponential. Hence, then the global runtime bounds can be exponential as well. Note that in the example program $\mathcal{P}'$ above, the eigenvalues of the update matrices of t_1 and t_4 have absolute value 1, i.e., t_1 and t_4 correspond to unit prs loops. Hence, by Theorem 45 we obtain polynomial runtime and size bounds for $\mathcal{P}'$.

Theorem 45 (Completeness Results for Integer Programs)

(a) Termination is decidable for all simple linear *integer programs where after chaining, all simple cycles correspond to prs loops.*
(b) Finite runtime and size bounds are computable for all simple integer programs where after chaining, all simple cycles correspond to terminating *prs loops.*
(c) If in addition to (b), all simple cycles correspond to unit *prs loops, then the runtime and size bounds are* polynomial.

In the definition of simple integer programs (Definition 44), we required that for every component $\mathcal{C}$ and every entry transition $(_, _, _, \ell) \in \mathcal{E}_{\mathcal{C}}$, there is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{*} (\ell, \sigma_0)$ for every $\sigma_0 \in \Sigma$. If one strengthens this by requiring that one can reach ℓ from ℓ_0 using only transitions whose guard is `true` and whose update is the identity, then the class of programs in Theorem 45 (a) is decidable (there are only n ways to chain a simple cycle with n transitions and checking whether a loop is a prs loop is decidable by Lemma 19).

6 Conclusion and Evaluation

Conclusion. In this paper, we developed techniques to infer size bounds automatically and to use them in order to obtain bounds on the runtime complexity of programs. This yields a complete procedure to prove termination and to infer

runtime and size bounds for a large class of integer programs. Moreover, we showed how to integrate the complete technique into an (incomplete) modular technique for general integer programs. To sum up, we presented the following new contributions in this paper:

(a) We showed how to use closed forms in order to infer size bounds for loops with possibly non-linear arithmetic in Theorem 7.
(b) We proved completeness of our novel approach for terminating prs loops (see Theorem 23) in Sect. 3.
(c) We embedded our approach for loops into the setting of general integer programs in Sect. 4 and showed completeness of our approach for simple integer programs with only prs loops in Sect. 5.
(d) Finally, we implemented a prototype of our procedure in our re-implementation of the tool KoAT, written in OCaml. It integrates the computation of size bounds via closed forms for twn-loops and homogeneous (and thus linear) solvable loops into the complexity analysis for general integer programs.[7]

To infer local runtime bounds as in Definition 41, KoAT first applies multiphase-linear ranking functions (see [5,19]), which can be done very efficiently. For twn-loops where no finite bound was found, it then uses the computability of runtime bounds for terminating twn-loops (see [17,20,27]). When computing size bounds, KoAT first applies the technique of [8] for reasons of efficiency and in case of exponential or infinite size bounds, it tries to compute size bounds via closed forms as in the current paper. Here, SymPy [30] is used to compute Jordan normal forms for the transformation to twn-loops. Moreover, KoAT applies a local control-flow refinement technique [19] (using the tool iRankFinder [13]) and preprocesses the program in the beginning, e.g., by extending the guards of transitions with invariants inferred by Apron [24]. For all SMT problems, KoAT uses Z3 [31]. In the future, we plan to extend the runtime bound inference of KoAT to prs loops and to extend our size bound computations also to suitable non-linear non-twn-loops.

Evaluation. To evaluate our new technique, we tested KoAT on the 504 benchmarks for *Complexity of* C *Integer Programs* (CINT) from the *Termination Problems Data Base* [35] which is used in the annual *Termination and Complexity Competition (TermComp)* [18]. Here, all variables are interpreted as integers over $\mathbb{Z}$ (i.e., without overflows). To distinguish the original version of KoAT [8] from our re-implementation, we refer to them as KoAT1 resp. KoAT2. We used the following configurations of KoAT2, which apply different techniques to infer size bounds.

- KoAT2orig uses the original technique from [8] to infer size bounds.
- KoAT2+SIZE additionally uses our novel approach with Theorem 7, 34, and 38.

[7] For a homogeneous solvable loop, the closed form of the twn-loop over $\mathbb{A}$ that results from its transformation is particularly easy to compute.

The CINT collection contains almost only examples with linear arithmetic and the existing tools can already solve most of its benchmarks which are not known to be non-terminating.[8] While most complexity analyzers are essentially restricted to programs with linear arithmetic, our new approach also succeeds on programs with *non-linear* arithmetic. Some programs with non-linear arithmetic could already be handled by KoAT due to our integration of the complete technique for the inference of local runtime bounds in [27]. But the approach from the current paper increases KoAT's power substantially for programs (possibly with non-linear arithmetic) where the values of variables computed in "earlier" loops influence the runtime of "later" loops (e.g., the modification of our example from Fig. 2 where t_4 decreases x_4 instead of x_1, see the end of Example 43).

Table 1. Evaluation on the Collection CINT^+

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \omega$	AVG$^+$(s)	AVG(s)
KoAT2+SIZE	26	233 (2)	71 (1)	25 (9)	3 (2)	358 (14)	9.97	22.88
KoAT2orig	26	232 (1)	70	15	5 (4)	348 (5)	8.29	21.52
MaxCore	23	220 (4)	67 (1)	7	0	317 (5)	1.96	5.25
CoFloCo	22	197 (1)	66	5	0	290 (1)	0.59	2.68
KoAT1	25	170 (1)	74	12	8 (3)	289 (4)	0.96	3.49
Loopus	17	171 (1)	50 (1)	6 (1)	0	244 (3)	0.40	0.40

Therefore, we extended CINT by 15 new typical benchmarks including the programs in (1), Fig. 2, and the modification of Fig. 2 discussed above, as well as several benchmarks from the literature (e.g., [3,6]), resulting in the collection CINT^+. For KoAT2 and KoAT1, we used Clang [11] and llvm2kittel [14] to transform C into integer programs as in Sect. 4. We compare KoAT2 with KoAT1 [8] and the tools CoFloCo [15], MaxCore [2] with CoFloCo in the backend, and Loopus [33]. These tools also rely on variants of size bounds: CoFloCo uses a set of constraints to measure the size of variables w.r.t. their initial and final values, MaxCore's size bound computations build upon [12], and Loopus considers suitable bounding invariants to infer size bounds.

Table 1 gives the results of our evaluation, where as in *TermComp*, we used a timeout of 5 min per example. The first entry in every cell denotes the number of benchmarks from CINT^+ for which the tool inferred the respective bound. The number in brackets only considers the 15 new examples. The runtime bounds computed by the tools are compared asymptotically as functions which depend on the largest initial absolute value n of all program variables. So for example, KoAT2+SIZE proved a linear runtime bound for $231 + 2 = 233$ benchmarks, i.e., $\mathrm{rc}(\sigma) \in \mathcal{O}(n)$ holds for all initial states where $|\sigma(v)| \leq n$ for all $v \in \mathcal{V}$.

[8] iRankFinder [13] proves non-termination for 119 programs in CINT. KoAT2orig already infers finite runtimes for 343 of the remaining $504 - 119 = 386$ examples in CINT.

Overall, this configuration succeeds on 358 examples, i.e., "$< \omega$" is the number of examples where a finite bound on the runtime complexity could be computed by the tool within the time limit. "AVG$^+$(s)" denotes the average runtime of successful runs in seconds, whereas "AVG(s)" is the average runtime of all runs.

Already on the original benchmarks CINT, integrating our novel technique for the inference of size bounds leads to the most powerful approach for runtime complexity analysis. The effect of the new size bound technique becomes even clearer when also considering our new examples which contain non-linear arithmetic and loops whose runtime depends on the results of earlier loops in the program. Thus, the new contributions of the paper are crucial in order to extend automated complexity analysis to larger programs with non-linear arithmetic.

KoAT's source code, a binary, and a Docker image are available at https://koat.verify.rwth-aachen.de/size. This website also has details on our experiments, a list and description of the new examples, and *web interfaces* to run KoAT's configurations directly online.

References

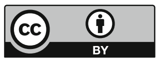
1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoret. Comput. Sci. **413**, 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009
2. Albert, E., Bofill, M., Borralleras, C., Martín-Martín, E., Rubio, A.: Resource analysis driven by (conditional) termination proofs. Theory Pract. Logic Program. **19**, 722–739 (2019). https://doi.org/10.1017/S1471068419000152
3. Ben-Amram, A.M., Jones, N.D., Kristiansen, L.: Linear, polynomial or exponential? Complexity inference in polynomial time. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 67–76. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_7
4. Ben-Amram, A.M., Pineles, A.: Flowchart programs, regular expressions, and decidability of polynomial growth-rate. In: Hamilton, G.W., Lisitsa, A., Nemytykh, A.P. (eds.) VPT 2016. EPTCS, vol. 216, pp. 24–49 (2016). https://doi.org/10.4204/EPTCS.216.2
5. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 601–620. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_32
6. Ben-Amram, A.M., Hamilton, G.W.: Tight worst-case bounds for polynomial loop programs. In: Bojańczyk, M., Simpson, A. (eds.) FoSSaCS 2019. LNCS, vol. 11425, pp. 80–97. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17127-8_5
7. Braverman, M.: Termination of integer linear programs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 372–385. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_34
8. 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
9. Cai, J.-Y.: Computing Jordan normal forms exactly for commuting matrices in polynomial time. Int. J. Found. Comput. Sci. **5**(3/4), 293–302 (1994). https://doi.org/10.1142/S0129054194000165

10. 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
11. Clang. Clang Compiler. https://clang.llvm.org/
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) POPL 1978, pp. 84–96 (1978). https://doi.org/10.1145/512760.512770
13. Doménech, J.J., Genaim, S.: iRankFinder. In: Lucas, S. (ed.) WST 2018, p. 83 (2018). https://wst2018.webs.upv.es/wst2018proceedings.pdf
14. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) RTA 2011. LIPIcs, vol. 10, pp. 41–50 (2011).https://doi.org/10.4230/LIPIcs.RTA.2011.41
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.: 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., Hark, M., Giesl, J.: Termination of polynomial loops. In: Pichardie, D., Sighireanu, M. (eds.) SAS 2020. LNCS, vol. 12389, pp. 89–112. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65474-0_5. https://arxiv.org/abs/1910.11588
18. 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
19. Giesl, J., Lommen, N., Hark, M., Meyer, F.: Improving automatic complexity analysis of integer programs. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) The Logic of Software: A Tasting Menu of Formal Methods. LNCS, vol. 13360, pp. 193–228. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08166-8_10
20. Hark, M., Frohn, F., Giesl, J.: Polynomial loops: beyond termination. In: Albert, E., Kovács, L. (eds.) LPAR 2020. EPiC, vol. 73, pp. 279–297 (2020). https://doi.org/10.29007/nxv1
21. 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
22. Hosseini, M., Ouaknine, J., Worrell, J.: Termination of linear loops over the integers. In: Baier, C., Chatzigiannakis, I., Flocchini, P., Leonardi, S. (eds.) ICALP 2019. LIPIcs, vol. 132 (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.118
23. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant generation for multi-path loops with polynomial assignments. In: Dillig, I., Palsberg, J. (eds.) VMCAI 2018. LNCS, vol. 10747, pp. 226–246. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_11
24. 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
25. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.: Closed forms for numerical loops. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019). https://doi.org/10.1145/3290368

26. Kovács, L.: Reasoning algebraically about P-solvable loops. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 249–264. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_18
27. Lommen, N., Meyer, F., Giesl, J.: Automatic complexity analysis of integer programs via triangular weakly non-linear loops. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 734–754. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_43
28. Lommen, N., Giesl, J.: Targeting completeness: using closed forms for size bounds of integer programs. CoRR, abs/2307.06921 (2023). https://doi.org/10.48550/arXiv.2307.06921
29. López-García, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. Theory Pract. Logic Program. **18**(2), 167–223 (2018). https://doi.org/10.1017/S1471068418000042
30. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3** (2017). https://doi.org/10.7717/peerj-cs.103
31. de Moura, L.M., 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
32. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial loop invariants: algebraic foundation. In: Gutierrez, J. (ed.) ISSAC 2004, pp. 266–273 (2004). https://doi.org/10.1145/1005285.1005324
33. 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
34. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_6
35. TPDB. (Termination Problems Data Base). https://github.com/TermCOMP/TPDB
36. Ming, X., Li, Z.-B.: Symbolic termination analysis of solvable loops. J. Symb. Comput. **50**, 28–49 (2013). https://doi.org/10.1016/j.jsc.2012.05.005

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.



Recurrence-Driven Summations in Automated Deduction

Visa Nummelin[1], Jasmin Blanchette[1,2(✉)], and Sander R. Dahmen[1]

[1] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
{visa.nummelin,s.r.dahmen,j.c.blanchette}@vu.nl
[2] Ludwig-Maximilians-Universität München, Munich, Germany
jasmin.blanchette@lmu.de

Abstract. Many problems in mathematics and computer science involve summations. We present a procedure that automatically proves equations involving finite summations, inspired by the theory of holonomic sequences. The procedure is designed to be interleaved with the activities of a higher-order automatic theorem prover. It performs an induction and automatically solves the induction step, leaving the base cases to the theorem prover.

1 Introduction

Finite summations—that is, summations $\sum_{i=m}^{n} t_i$ over finitely many terms t_i—are ubiquitous in mathematics and computer science, but they are poorly supported by automatic theorem provers. One reason is that summations are higher-order, whereas most theorem provers are first-order.

In recent years, we have seen the rise of higher-order provers [2,3,16–18]. With these provers, $\sum_{i=m}^{n} t_i$ can be represented as $\mathsf{sum}\ m\ n\ (\lambda i.\ t_i)$; the traditional $\sum$ syntax can be seen as syntactic sugar. But despite the use of heuristics [17, Sect. 4], higher-order provers are ill-equipped to reason inductively. A simple problem such as $\sum_{i=0}^{\mathsf{n}} i = \mathsf{n}(\mathsf{n}+1)/2$ is a formidable challenge for them, even if we include axioms for $+$, $\cdot$, $/$, and $\sum$ together with an induction principle.

In this paper, we introduce a procedure for proving such equations in a higher-order prover. The procedure is triggered by a proof goal of the form $k\sum s + t = u$, possibly with some conditions (Sect. 2). In a refutational prover, the equation would be negated, as $k\sum s + t \neq u$, and would correspond to the negated conjecture, a problem axiom, or some clause derived by the prover.

Our procedure translates facts about summations to linear recurrences. These recurrences have almost the same form as multivariate holonomic sequences [20], which, while not being a prerequisite for reading this paper, strongly inspired our work. Each recurrence is associated with a multivariate sequence—a sequence with one or more indices. In this paper, the word "sequence" generally means "multivariate sequence."

The procedure has three steps.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 23–40, 2023.
https://doi.org/10.1007/978-3-031-43369-6_2

1. *Initialization* (Sect. 3): Heuristically choose terms in the goal to generalize and perform induction on. Among the problem axioms, select those of a suitable form as initial recurrences for the procedure.
2. *Propagation* (Sect. 4): From the initial recurrences, compute recurrences corresponding to the goal. For $+$, $\cdot$, and $\sum$ expressions occurring in the goal, recurrences are computed from the recurrences of their direct subexpressions.
3. *Induction* (Sect. 5): If the final recurrences for the goal involve only the goal and no other sequences, use them for induction. If they make the difference of successive values of $k\sum s+t-u$ constantly 0, this establishes the induction step. Then reduce the goal to a set of base cases and give these to the prover.

Propagation and induction apply holonomic-style techniques almost as a black box. Initialization connects them to the overall proof search.

For example, to prove $\sum_{i=0}^{\mathsf{n}} i = \mathsf{n}(\mathsf{n}+1)/2$, the procedure would transform the equation into recurrences and find out that the difference $\sum_{i=0}^{\mathsf{n}} i - \mathsf{n}(\mathsf{n}+1)/2$ remains constant as n increases, thereby establishing the induction step. If that difference is constantly 0, we get $\sum_{i=0}^{\mathsf{n}} i = \mathsf{n}(\mathsf{n}+1)/2$; in general, it suffices to prove a number of base cases, which are left to the prover. This example is very simple, but the procedure scales up to more sophisticated problems (Sect. 6). An implementation is under way in the Zipperposition prover [17].

The procedure treats $\sum$ as an interpreted (built-in) symbol. The summation expression evaluates to a value in a commutative group, or a ring if ring multiplication is present. The commutative group or ring gives us $+$, $\cdot$, and $-$. These are also interpreted, as are numerals. Integers, including indices, can multiply group elements. Based on the interpretation, we use the forms $t = u$ and $t - u = 0$ interchangeably.

Compared with Wilf–Zeilberger pairs [19] and other methods (Sect. 7), the main benefit of our procedure is that it goes beyond holonomic sequences and supports both uninterpreted functions and an infinite number of base cases. Our procedure is widely applicable and may help prove not only difficult summations in a restrictive form but also easier summations in a more general form, which is useful in a general-purpose theorem prover. At the heart of our work is the novel combination of techniques from superposition and holonomic sequences, which is visible both in the prover integration (Sect. 2) and in the computation of so-called excess terms (Sect. 4). We refer to our technical report [14] for more details.

2 Inference Rule

Our procedure can be integrated into a theorem prover, where it takes the form of an inference rule that complements the prover's existing rules. Our technical report discusses an integration with satisfiability modulo theories (SMT) and tableaux; here, we present a rule for superposition:

$$\frac{C_1 \ \cdots \ C_l \quad C' \vee t[\vec{s}] \neq 0}{D \vee C' \vee \bigvee_{\vec{b} \in B} t[\vec{b}] \neq 0}\ \textsc{Summation}$$

These side conditions apply:

- $t[\vec{s}]$ is an expression that can be brought into the general form $k\sum_{i=m}^{n} t' + t''$;
- the procedure selects, generalizes, and performs an induction on the subterms $\vec{s}$ of t (Sect. 3);
- the procedure succeeds at proving the induction step based on initial recurrences derived from $C_1, \ldots, C_l$ (Sect. 3) and their propagation (Sect. 4);
- the procedure identifies B as the finite set of base cases of the induction, where each case is a vector $\vec{b}$ of terms of the same length as $\vec{s}$ (Sect. 5); and
- the subclause D captures potential conditions determined by the procedure.

The intuition behind the rule is that the conclusion should be easier to refute than the rightmost premise. As for the premises $C_1, \ldots, C_l$, they can contain useful information about $\vec{s}$, often about bounds.

3 Initialization

The first step of our procedure is to recognize the structure of recurrences. Variables on which we can perform induction appear as Skolem constants in the negated goal. Further opportunities for induction can be created by generalizing complex terms. Also as part of this step, we must choose which terms represent (multivariate) sequences and which clauses represent their recurrences.

Theory Detection. We require the necessary theory of summation to be predefined. Specifically, this refers to the inductive theory of integers, axioms for commutative groups (including multiplication by integers), and the definition of summation from 0 by $\sum_{n=0}^{-1} f_n = 0$ and $\sum_{n=0}^{m+1} f_n = \sum_{n=0}^{m} f_n + f_{m+1}$ even for negative $m \in \mathbb{Z}$. Other finite intervals than $[0, m]$ are expressible as differences.

Ring multiplication may be absent, so we do not take it as predefined. Instead, we search candidate binary operators from the negated goal. For each candidate, we can try to prove left and right distributivity by syntactically looking for that axiom or by running another instance of the prover. Distributivity is the only necessary property to apply the procedure, but associativity, commutativity, and the unit element can also be used in simplifications.

Term Generalization. Term generalization transforms Skolem constants or complex terms into variables and then performs an induction on the variables. We propose a straightforward heuristic: For each nonnumeral subterm s of type $\mathbb{Z}$ occurring in the negated goal, generalize s if s stays variable-free even after recursively applying this heuristic on the proper subterms of s itself. For example, in the following variable-free integer terms, the underlined subterms would be generalized: $\underline{\mathsf{a}}$, 123, $\underline{\mathsf{f}\ 0\ 2}$, $2\mathsf{f}\ (\underline{\mathsf{g}\ (-1)})\ (-3\underline{\mathsf{a}})$, $\mathsf{f}\ 1\ (\mathsf{g}\ (\underline{\mathsf{a}} + 1))$, $\mathsf{f}\ (\mathsf{g}\ \underline{\mathsf{a}})\ 7\underline{\mathsf{a}}$.

Let $\vec{s} = (s_1, \ldots, s_d)$ be the subterms chosen for generalization. Then, based on the negated goal $C' \vee t[\vec{s}] \neq 0$ (as in the Summation rule), generalization sets up the goal $\forall \vec{n} \in N.\ t[\vec{n}] = 0$ where $N \subseteq \mathbb{Z}^d$ collects the bounds of $\vec{s}$ (often $N = \mathbb{N}^d$). We try to prove this goal up to base cases and other mild conditions.

The generalization makes it possible to use induction to prove that the goal sequence term $t[\vec{n}]$—a function of $\vec{n}$—equals zero on N. We try to prove the generalized goal assuming $\neg C'$ and some extra conditions E such as the base cases of the induction. Then, instantiating $\vec{n} := \vec{s}$, we conclude $C' \vee \neg E \vee t[\vec{s}] = 0$. This, together with the negated goal $C' \vee t[\vec{s}] \neq 0$, implies a conclusion of the form $C' \vee \neg E$ for the SUMMATION rule. Note that C' is not generalized.

The set N embodies knowledge about $\vec{s}$ that we find among existing clauses $C_1, \ldots, C_l$ and the condition $\neg C'$. The free variables of $\neg C'$ are interpreted as constants, and they can also occur in $\vec{s}$. For example, assume that $\vec{s} = \mathsf{f}\ s'$ and $\vec{n} = n$ and that the generalized goal contains the factorial $n!$. Its recurrence must be in a conditional clause—e.g., $(m+1)! = (m+1)\, m! \vee 0 \nleq m$. To use this recurrence for $n!$, we need $n \geq 0$, which we can ensure using N if we find a bounding clause $\mathsf{f}\ s' \geq 0$ or its generalization such as $\mathsf{f}\ m \geq 0$ where m is a free variable. The more we know about $\vec{s}$, the more recurrences we can get. At the same time, N must allow induction, so we keep it convex by considering only coordinatewise bounds of $\vec{s}$.

Form of Sequence Terms. Sequence terms are terms of the underlying higher-order logic that our procedure can work with. From their structure, we distinguish (pointwise) addition and multiplication, summation, and affine substitution. This gives a first-order grammar to express the sequence terms.

Definition 1. *Sequence terms* on a ring A are inductively defined as follows. The logic's terms of type A with distinguished integer variables $\vec{n}$ are sequence terms. If $f_{\vec{n}}$ and $g_{\vec{n}}$ are sequence terms with d variables $\vec{n}$, then so are $f_{\vec{n}} + g_{\vec{n}}$, $f_{\vec{n}} \cdot g_{\vec{n}}$, $\sum_{i=0}^{\vec{c} \bullet \vec{n} + a} f_{\{n_j \mapsto i\}\vec{n}}$, and $\sigma f_{\vec{n}} = f_{\sigma \vec{n}}$ where $\vec{c}$ is a vector, a is an integer, $\vec{c} \bullet \vec{n} = c_1 n_1 + \cdots + c_d n_d$, and σ is an affine substitution (meaning $\sigma \vec{m} = q\vec{m} + \vec{b}$ for a matrix q and a vector $\vec{b}$); a, the entries of $\vec{c}$, and the entries of σ (meaning the entries of q and $\vec{b}$) must be numerals.

Remark 2. In Definition 1 and in the sequel, a commutative group can be used instead of a ring if ring multiplication is absent. In this case, all formulas involving ring multiplication (e.g., $f_{\vec{n}} \cdot g_{\vec{n}}$) should be ignored.

We view sequence terms as functions $\mathbb{Z}^d \to A$. We then write the sequence terms from the definition compactly as $f + g$, $f \cdot g$, $\sum_j^a f$, and σf, and call $a = \vec{c} \bullet \vec{n} + d$ an affine variable sum. Moreover, since $\cdot$, $\sum_j^a$, and σ all distribute over $+$, we can write any sequence term as $c_1 f^1 + \cdots + c_k f^k$ where the coefficients c_j are numerals and the sequence terms f^j are distinct and do not contain $+$. Finally, we forbid variable shadowing: $\sum_{n_j=0}^a$ binds n_j, and while $\sum_j^a \sum_j^b g$ and $\sum_j^{n_j} g$ and other references to n_j outside $\sum_j^a$ are syntactically valid, we avoid such forms by renaming them during encoding and never reintroducing them.

Choice of Initial Recurrences. Semantically, the recurrences we look for are multivariate heterogeneous linear finite-fixed-step equations with polynomial coefficients. An archetypical example is

$$\left(n^2+1\right) f_{n+2,m+1} + m f_{n+1,m} = nm f_{n,m+1} - 2h_{n,m} + (m-n)\, h_{n,m+1} + 1 \quad (1)$$

Here, the sequences $f, h, 1$ are bivariate, and the sequence indices are all of the form $n + k$ or $m + k$ for numerals $k \in \mathbb{Z}$, amounting to finite fixed steps.

The general form is $0 = P_1 g^1 + \cdots + P_k g^k = \vec{P} \bullet \vec{g}$ where $\vec{g} = \left(g^1, ..., g^k\right)$ is a tuple of sequence terms and $\vec{P}$ is a tuple of operator polynomials as defined below. If $k = 1$, we have a homogeneous recurrence of g^1; otherwise, it is heterogeneous.

Definition 3. *Operator polynomials* are a $\mathbb{Z}$-algebra with composition as product (meaning closed under addition, composition, and integer multiplication) spanned by the multiplier and shift operators:

- The *multiplier* operator M_j of index j multiplies a multivariate sequence f by the variable n_j of index j: $(M_j f)_{\vec{n}} = n_j f_{\vec{n}}$.
- The *shift* operator $S_j = \{n_j \mapsto n_j + 1\}$ of index j increments the variable n_j of index j of a multivariate sequence f by one: $(S_j f)_{\vec{n}} = f_{\{n_j \mapsto n_j+1\}\vec{n}}$.

With d index variables, the operator polynomials look like ordinary polynomials $\mathbb{Z}[M_1, ..., M_d, S_1, ..., S_d]$, but the composition product is noncommutative since $S_i M_i = M_i S_i + S_i$ for all $i = 1, \ldots, d$ (a derivation of which is given in the next section directly above equation (2)). As an example of expressing recurrences in terms of operator polynomials, consider the previous archetypical recurrence (1). Taking n as the first and m as the second variable, the recurrence reads

$$((M_1^2 + 1)S_1^2 S_2 + M_2 S_1 - M_1 M_2 S_2) \cdot f + (2 - (M_2 - M_1)S_2) \cdot h + (-1) \cdot 1 = 0$$

Remark 4. The expression $\vec{P} \bullet \vec{g}$ identifying a recurrence is itself a sequence term. It suffices to observe that if f is a sequence term, then so are the substitution $S_j f$ and the product $M_j f = (\vec{n} \mapsto n_j) \cdot f$ with the projection sequence term $\vec{n} \mapsto n_j$.

As sketched in Sect. 1, we must select some of the problem axioms as initial recurrences for the procedure. This is accomplished as follows. Let there be an edge between two axioms of the form $C \vee s = t$ (where C may be empty) if they both contain a top-level occurrence of the same sequence g, i.e., an occurrence of g that is not nested inside an uninterpreted function symbol. The axioms then form a graph. We take as initial recurrences the connected component of the generalized goal.

By a sequence g, we mean the $\mathsf{f}\ \vec{a}$ part of a term of the form $\mathsf{f}\ \vec{a}\ \vec{n}$ where f is an uninterpreted function symbol, $\vec{a}$ is a tuple of variable-free terms, and $\vec{n}$ is a nonempty tuple of integer variables or affine (i.e., linear term + constant term) combinations of them. The tuples $\vec{a}$ and $\vec{n}$ may in general be interleaved.

In other contexts, an analogous step is known as lemma filtering or premise selection [4, Sect. 2]. Clutter from irrelevant facts is less of an issue in the context of our procedure because it can use only linear recurrences. Beyond this, our simple heuristic does nothing to avoid clutter.

What should we do about conditions such as C in $C \vee \mathsf{f}\ \vec{a}\ \vec{n} = t$? We could forbid them and work only with unit equations such as $\mathsf{f}\ \vec{a}\ \vec{n} = t$. We could collect

them and put them in the D component of the Summation rule's conclusion. Or we could attempt to prove them when the initial recurrences are selected. In our ongoing implementation, we chose the first option, but what the best option is remains an open question.

4 Propagation

Holonomic sequences can be defined by homogeneous recurrences with polynomial coefficients and finitely many base cases. They are closed under the four operations that build sequence terms $(+, \cdot, \sum_j^a, \sigma)$, which especially makes their equality decidable [20]. The closure is realized by four procedures to derive recurrences of a sequence term from the recurrences of its immediate subterms, which we call *propagation*. We can propagate independently of the base cases and hence work on nonholonomic sequence terms [6]. Although we expect the holonomic subcase to be decidable in our setting, in general decidable equality is lost. Additionally, unlike in the holonomic setting, we allow heterogeneous recurrences. We will build this into our noncommutative Gröbner basis setup that is used in the propagation procedures.

Gröbner Bases of Recurrence Operators. A (generalized) Gröbner basis is a certain well-behaved generating set of a left-ideal of (possibly noncommutative) polynomials. Equivalently, we will view it as a system of polynomial equations that is complete for rewriting. Given a polynomial equation $P = 0$, for every monomial M we get a rewrite rule as follows. Decompose MP as $MP = L + R$ where L is the leading monomial of MP w.r.t. a fixed monomial ordering times its coefficient. Then $L = -R$ gives rise to a rewrite rule $L \to -R$. A system of equations is complete for rewriting if every one of its consequences can be proved via rewriting by these rules.

Example 5. The system $\{ab^2 = a + b,\, a^2b = a + 1\}$ does not prove its consequence $a^2 = b$ by rewriting. (We can see that $a^2 = b$ is a consequence by multiplying the first equation by a and the second equation by b and then by subtracting the two equations.) In the other direction, the system's Gröbner basis $\{a^2 = b,\, b^2 = a + 1\}$ does give rewrite proofs $ab^2 \xrightarrow[b^2=a+1]{} a^2 + a \xrightarrow[a^2=b]{} b + a$ and $a^2b \xrightarrow[a^2=b]{} b^2 \xrightarrow[b^2=a+1]{} a + 1$.

A theory of Gröbner bases exists for various polynomial algebras [10]. In our setting, a sufficient requirement is that all indeterminates X, Y commute up to lower-order terms: $XY - YX \in \mathbb{Z}X + \mathbb{Z}Y + \mathbb{Z}$. The operator polynomials of Definition 3 fall into this category with the natural choice of taking all multiplier and shift operators as indeterminates. Indeed, for any sequence term f, we have the noncommutation relations

$$(S_j M_j f)_{\vec{n}} = (S_j\,(\vec{n} \mapsto n_j f_{\vec{n}}))_{\vec{n}} = (n_j + 1)\,(S_j f)_{\vec{n}} = ((M_j S_j + S_j)\,f)_{\vec{n}}$$

and all other pairs of multipliers and shifts commute exactly. That is:

$$S_i M_j = M_j S_i + \delta_{i,j} S_i \qquad S_i S_j = S_j S_i \qquad M_i M_j = M_j M_i \tag{2}$$

for all i and j, where $\delta_{i,j}$ equals 1 if $i = j$ and 0 otherwise. When we consider a formal polynomial algebra (necessary to perform Gröbner basis computations), we will usually mean polynomials with integer coefficients and indeterminates $M_1, M_2, \ldots, S_1, S_2, \ldots$ satisfying (2). Exceptionally, when we use propagation to substitution, we will consider compositions of shifts formally as further individual indeterminates, as explained above Procedure 12. Apart from this exceptional setting, we fix a choice of monomials as follows.

Definition 6. In our setting, a *monomial* is a polynomial of the form $M_1^{x_1} \cdots M_d^{x_d} S_1^{y_1} \cdots S_d^{y_d}$ where the exponents $x_j, y_j \in \mathbb{N}$ are numerals.

Due to the (non)commutation relations (2), polynomials can be written as sums of monomials times their integer coefficients. This makes working with these noncommutative polynomials similar to working with commutative ones. A major difference is that monomials are not closed under product, as illustrated by $S_1 \cdot M_1 = M_1 S_1 + S_1$. This complicates the definition of monomial order below, which in turn defines how to interpret a polynomial equation as a rewrite rule.

Definition 7. A *monomial order* $\preccurlyeq$ is a well-founded total order on monomials such that for all monomials A, B, C, if $A \preccurlyeq B$, then the leading monomial of CA is $\preccurlyeq$-smaller than the leading monomial of CB; here, the *leading monomial* of a nonzero polynomial P means the $\preccurlyeq$-largest monomial occurring in P.

Buchberger's algorithm to compute Gröbner bases (also in a noncommutative context) is similar to saturation-based theorem proving. It repeatedly derives from polynomial equations $P = 0$ and $R = 0$ new equations $AP - BR = 0$ where coefficient–monomial products A, B make the leading monomials of AP and BR cancel. It suffices to take A, B with smallest total degree and coprime coefficient. A and B play a similar role to the most general unifier in superposition. Since S_j is semantically bijective, we can and always do cancel it, replacing $S_j R = 0$ by $R = 0$. This modified completion into a Gröbner basis always terminates. The standard termination proof reduces to applying noetherianity of commutative polynomials over $\mathbb{Z}$ or Dickson's lemma [10].

A single operator polynomial P_1 perfectly encodes a linear homogeneous recurrence $0 = P_1 g$ of a sequence term g. However, we allow any heterogeneous recurrence of the form $0 = \vec{P} \bullet \vec{f} = P_1 f^1 + \cdots + P_k f^k$ where $\vec{f} = \left(f^1, \ldots, f^k\right)$ is an arbitrary tuple of different sequence terms. We can encode this by a single operator polynomial for the duration of one Gröbner basis computation as follows. Let $\vec{f}$ enumerate exactly once all the sequence terms needed to express the current recurrences with the help of operator polynomials. Let $\vec{f}$ depend on d variables. For each f^j, we consider a shift $F_j := S_{d+j}$ w.r.t. a so far unused variable. Then the operator polynomial $\vec{P} \bullet \vec{F}$ encodes $0 = \vec{P} \bullet \vec{f}$.

This encoding does not respect the semantics of operator polynomials; to recover it, we must apply the substitution $\{\vec{F} \mapsto \vec{f}\}$. However, products such as $F_1 F_2$ remain uninterpretable even with ring-valued sequences because the operator product—function composition—is different from multiplication of f^j's. Hence, we will simply discard uninterpretable polynomials after the Gröbner basis computation. Moreover, from now on, we will freely write f^j for F_j.

Definition 8. Let $X_1, \dots, X_n$ be an enumeration of all multiplier and shift indeterminates. An $(X_1, \dots, X_k)$-*elimination order* is a monomial order such that $X_j \succ X_{k+1}^{a_{k+1}} \cdots X_n^{a_n}$ for all indices $j \le k$ and all exponents $a_{k+1}, \dots, a_n \in \mathbb{N}$.

Our default choice for the order is to compare total degree in $X_1, ..., X_k$ and break ties using the total degree reverse lexicographical order [7, Chapter 2 §2].

Procedure 9. *Eliminating* indeterminates $X_1, \dots, X_k$ from a finite system of equations E means computing a Gröbner basis G of E w.r.t. an $(X_1, \dots, X_k)$-elimination order and then discarding all polynomials from G that contain any of $X_1, \dots, X_k$ or that are not linear in the indeterminates encoding sequence terms. (As mentioned above, during the Gröbner basis computation, whenever we derive a polynomial $S_j R$, we replace it by R.)

While in principle any Gröbner basis would suffice for elimination, our default choice is to compute the reduced Gröbner basis (i.e., the fully simplified one). The nonlinear polynomials can be discarded as soon as they are derived during the Gröbner basis computation instead of only at the end. Recurrence equations produced by elimination are logical consequences of the input equations, as we explain in our technical report.

Despite the formally equivalent roles of all sequence terms f^i in the recurrence $0 = \vec{P} \bullet \vec{f}$, we associate with every recurrence a sequence term f^j. It is often convenient to write such a recurrence of f^j as $P_j f^j + e = 0$ where the *excess terms* $e = \vec{P} \bullet \vec{f} - P_j f^j$ contain all sequence terms f^i except f^j. The choice of f^j among $\vec{f}$ will be determined by the definition of excess terms (Definition 18). However, this choice remains irrelevant for the individual propagation steps, described below. We adapt these steps from the four closure properties of holonomic sequences by carrying excess terms along.

Propagation to Addition. Let us start with addition of sequence terms.

Procedure 10. Let f and g be sequence terms, and let h be the formal name of their addition $f+g$. The associated recurrences F of f and G of g are propagated to those of h by eliminating f and g from $F \cup G \cup \{h = f + g\}$. (By Procedure 9, this involves computing a Gröbner basis for these equations and then discarding the equations containing f or g as well as the corresponding nonlinear terms.)

Actually, the same propagation technique works if $f + g$ is replaced by any expression in the general recurrence format $\vec{P} \bullet \vec{l}$ (a dot product of operator polynomials $\vec{P}$ and sequence terms $\vec{l}$). The key is that the defining equation $h = \vec{P} \bullet \vec{l}$ is again a linear recurrence. Such propagations could also be done by iterating more primitive propagations.

Example 11. Consider the goal $\sum_{j=0}^{n} a_j = g_n + a_0$ given $g_0 = 0$ and $g_{n+2} = g_n + a_{n+1} + a_{n+2}$ for all $n \in \mathbb{N}$. The defining recurrence of g can be written using the operator polynomials as $S_1^2 g = g + S_1 a + S_1^2 a$. The defining recurrence of the sum $f_n := \sum_{j=0}^{n} a_j$ is $S_1 f = f + S_1 a$. We must prove that $h_n := g_n + a_0 - f_n$

is 0. To achieve this, we propagate recurrences to h using the elimination procedure described above (Procedure 9) and the total-degree-based (f, g)-elimination order with $f \prec g$. Leading monomials are shown in bold:

$$
\begin{array}{lll}
 & 0 = \boldsymbol{S_1^2 g} - g - S_1 a - S_1^2 a & \text{recurrence of } g \\
-S_1^2 & 0 = \boldsymbol{g} + a_0 - f - h & \text{definition of } h \\
\hline
 & 0 = -g - S_1 a - S_1^2 a - a_0 + \boldsymbol{S_1^2 f} + S_1^2 h & \\
-S_1 & 0 = \boldsymbol{S_1 f} - f - S_1 a & \text{recurrence of } f \\
\hline
 & 0 = -g - S_1 a - a_0 + S_1^2 h + \boldsymbol{S_1 f} & \\
- & 0 = \boldsymbol{S_1 f} - f - S_1 a & \text{recurrence of } f \\
\hline
 & 0 = -\boldsymbol{g} - a_0 + S_1^2 h + f & \\
+ & 0 = \boldsymbol{g} + a_0 - f - h & \text{definition of } h \\
\hline
 & 0 = \boldsymbol{S_1^2 h} - h &
\end{array}
$$

In this example, $h_{n+2} - h_n = 0$ is the only recurrence that does not contain f and g, so we discard the rest of the Gröbner basis calculation. Since $h_{n+2} - h_n = 0$ contains only the sequence h, we can use it to prove the induction step (of size 2) of a proof of $\forall n.\ h_n = 0$. We are then left with the two base cases $h_0 = 0$ and $h_1 = 0$, which the SUMMATION inference would include in its conclusion without auxiliary symbols (f and h) as $\sum_{j=0}^{0} a_j \neq g_0 + a_0 \vee \sum_{j=0}^{1} a_j \neq g_1 + a_0$.

Propagation to Substitution. Consider a numeral matrix $a = [a_{kj}]_{kj} \in \mathbb{Z}^{d \times D}$ and a vector $\vec{b} \in \mathbb{Z}^d$. They characterize an affine substitution $\sigma = \{\vec{n} \mapsto a\vec{n} + \vec{b}\} = \{n_k \mapsto \sum_{j=1}^{D} a_{kj} n_j + b_k \mid 1 \le k \le d\}$. As an operator on sequences, σ performs an affine change of variables: $(\sigma f)_{\vec{n}} = f_{a\vec{n}+\vec{b}}$.

Clearly, any recurrence $Pf = 0$ of f implies $\sigma P f = 0$. Moreover, if $\sigma P = P'\sigma$, then $P'\sigma f = 0$ gives a recurrence of σf. Finding such a P' for a general P can be reduced to finding an operator polynomial P'_X satisfying $\sigma X = P'_X \sigma$ for every indeterminant X. This amounts to pushing all indeterminates X leftwards. For multipliers, we have $\sigma(M_1, \ldots, M_d) = (a(M_1, \ldots, M_D) + \vec{b})\,\sigma$. In contrast, shifts are easily pushed only rightwards—namely, $S_j \sigma = \sigma S_1^{a_{1j}} \cdots S_d^{a_{dj}}$. Consequently, the recurrences of f must be first expressed in terms of the composite shifts $\mathbb{S}_j := S_1^{a_{1j}} \cdots S_d^{a_{dj}}$. As operators, these satisfy the (non)commutation relations

$$
\mathbb{S}_j M_k = (M_k + a_{kj})\,\mathbb{S}_j \qquad \mathbb{S}_i \mathbb{S}_j = \mathbb{S}_j \mathbb{S}_i \qquad \mathbb{S}_i S_j = S_j \mathbb{S}_i \tag{3}
$$

This makes the $\mathbb{S}_j$'s suitable as indeterminates in Gröbner basis computations.

Accordingly, for propagation to substitution, we enlarge our formal polynomial algebra to also contain the indeterminates $\mathbb{S}_1, \mathbb{S}_2, \ldots$ satisfying the relations (3), while also keeping (2). We note that, as operators, the indeterminates further satisfy (essentially by definition) the relations

$$
\mathbb{S}_j \prod_{k:\, a_{kj} < 0} S_k^{|a_{kj}|} = \prod_{k:\, a_{kj} > 0} S_k^{a_{kj}} \quad \text{for } j \in \{1, \ldots, D\} \tag{4}
$$

We add these new relations to the system of recurrence equations of which we compute the Gröbner basis. Finally, we extend our notion of *monomial* from Definition 6 to mean any polynomial of the form $M_1^{x_1} \cdots M_d^{x_d} S_1^{y_1} \cdots S_d^{y_d} \mathbb{S}_1^{z_1} \cdots \mathbb{S}_D^{z_D}$ where the exponents $x_j, y_j, z_j \in \mathbb{N}$ are numerals.

Procedure 12. Recurrences of a sequence term f are propagated to its affine substitution $(\sigma f)_{\vec{n}} = f_{a\vec{n}+\vec{b}}$ as follows. Eliminate each S_k from the system of polynomial equations containing both the recurrences of f and the relations (4). Every resulting recurrence $P(\vec{M}, \vec{\mathbb{S}})f + e = 0$ implies a recurrence $P(a\vec{M} + \vec{b}, \vec{S})\sigma f + \sigma e = 0$ of σf where we have collected the indeterminates into vectors and where e are excess terms that do not contain f.

Example 13. Consider $\sum_{n_1=0}^{n_2} \binom{n_1}{n_2-n_1} = F_{n_2+1}$ where the Fibonacci numbers are defined by $F_1 = F_2 = 1$ and $\left(S_1^2 - S_1 - 1\right) F = 0$. For the binomial coefficient $\binom{\cdot}{\cdot}_{n_1,n_2} = \binom{n_1}{n_2} = \frac{n_1!}{n_2!(n_1-n_2)!}$, the recurrence from Pascal's triangle reads as $(S_1 S_2 - S_2 - 1)\binom{\cdot}{\cdot} = 0$ and extends $\binom{n_1}{n_2}$ from $0 \leq n_2 \leq n_1$ to all $n_2 \in \mathbb{Z}$ and $n_1 \in \mathbb{N}$. Moreover, we have $\binom{n_1}{n_2} = \frac{n_1}{n_2}\binom{n_1-1}{n_2-1}$—i.e., $((M_2+1) S_1 S_2 - M_1 - 1)\binom{\cdot}{\cdot} = 0$. We want to propagate these recurrences to the substitution $\sigma = \{n_1 \mapsto n_1,\ n_2 \mapsto n_2 - n_1\}$. We have $S_1 \sigma = \sigma S_1 S_2^{-1}$ and $S_2 \sigma = \sigma S_2$. So we introduce for $S_1 S_2^{-1}$ and S_2 the indeterminates $\mathbb{S}_1$ and $\mathbb{S}_2$ whose characterizing recurrences (4) read

$$(\mathbb{S}_1 S_2 - S_1)\binom{\cdot}{\cdot} = 0 \quad \text{(i)} \qquad\qquad (\mathbb{S}_2 - S_2)\binom{\cdot}{\cdot} = 0 \quad \text{(ii)}$$

Next, we eliminate S_1, S_2 in favor of $\mathbb{S}_1, \mathbb{S}_2$. Here, (ii) immediately rewrites every S_2 to $\mathbb{S}_2$ and then (i) becomes $(-S_1 + \mathbb{S}_1\mathbb{S}_2)\binom{\cdot}{\cdot} = 0$, which rewrites every S_1. The remaining steps to complete a Gröbner basis w.r.t. some total-degree order are irrelevant for what we want to illustrate. We factor the result for readability:

$$(-S_1 + \mathbb{S}_1\mathbb{S}_2)\binom{\cdot}{\cdot} = 0 \qquad (-S_2 + \mathbb{S}_2)\binom{\cdot}{\cdot} = 0$$
$$\left(\mathbb{S}_1\mathbb{S}_2^2 - \mathbb{S}_2 - 1\right)\binom{\cdot}{\cdot} = 0 \qquad ((M_2+1)\,\mathbb{S}_2 - M_1 + M_2)\binom{\cdot}{\cdot} = 0$$
$$((M_1+1)\,\mathbb{S}_1\mathbb{S}_2 - (M_1 - M_2 + 2)\,\mathbb{S}_1 - M_1 - 1)\binom{\cdot}{\cdot} = 0$$
$$((M_1 - M_2 + 1)\,(M_1 - M_2 + 2)\,\mathbb{S}_1 - M_1 M_2 - M_2)\binom{\cdot}{\cdot} = 0$$

Now σ maps the lowest four recurrences to recurrences of $f_{n_1,n_2} = \binom{n_1}{n_2-n_1}$ below:

$$\left(S_1 S_2^2 - S_2 - 1\right) f = 0 \qquad ((M_2 - M_1 + 1)\,S_2 - 2M_1 + M_2)\,f = 0$$
$$((M_1+1)\,S_1 S_2 - (2M_1 - M_2 + 2)\,S_1 - M_1 - 1)f = 0$$
$$((2M_1 - M_2 + 1)\,(2M_1 - M_2 + 2)\,S_1 - (M_1+1)\,(M_2 - M_1))f = 0$$

The next step is to propagate to the summation. We postpone it to Example 16.

Propagation to Product. Let $\cdot$ be ring multiplication or more generally a group bihomomorphism. If the sequence terms f and g depend on disjoint sets of variables, recurrences of $fg = f \cdot g$ are essentially a union of recurrences of f and g. Namely, let $Pf + e = 0$ be any recurrence of f where P is an operator polynomial on the variables of f and the excess terms e do not contain f. Then $P(fg) + eg = 0$ because g is effectively a constant to P, and similarly for recurrences of g. With the help of this special case, propagation to product can be reduced to propagation to substitution, as explained below.

Procedure 14. Let f and g be sequence terms parameterized by the variables $\vec{n} = (n_j)_{j=1}^d$. Let $\vec{m} = (n_{j+d})_{j=1}^d$ be a tuple of fresh variables. The recurrences of f and g are propagated to their pointwise product fg in two steps. First, the recurrences of the variable-disjoint product $f_{\vec{n}} g_{\vec{m}}$ are the union of the recurrences of $f_{\vec{n}}$ multiplied on the right by $g_{\vec{m}}$ and of those of $g_{\vec{m}}$ multiplied on the left by $f_{\vec{n}}$. Then the recurrences of $f_{\vec{n}} g_{\vec{n}} = \{\vec{m} \mapsto \vec{n}\}(f_{\vec{n}} g_{\vec{m}})$ are found by propagating to substitution using Procedure 12.

Propagation to Summation. We finally consider the summations $\sum_{n_1=0}^{n_2} f_{\vec{n}}$. We can assume that the variables are numbered so that the sum acts on the first two. Similarly to above, we consider the consequence $\sum_{n_1=0}^{n_2} P f_{\vec{n}} + \sum_{n_1=0}^{n_2} e_{\vec{n}} = 0$ of a recurrence $Pf + e = 0$ of the sequence term f where P is an operator polynomial and e are excess terms. We want to find an operator polynomial P' such that $\sum_{n_1=0}^{n_2} P$ becomes $P' \sum_{n_1=0}^{n_2}$ up to excess terms. Like for substitutions, finding such a P' for P can be reduced to finding an operator polynomial P'_X satisfying $\sum_{n_1=0}^{n_2} X = P'_X \sum_{n_1=0}^{n_2}$ up to excess terms for every indeterminant X. The result will be a recurrence $P' \sum_{n_1=0}^{n_2} f_{\vec{n}} + e' = 0$ of $\sum_{n_1=0}^{n_2} f_{\vec{n}}$.

Procedure 15. Recurrences of a sequence term f are propagated to its sum $\sum_{n_1=0}^{n_2} f_{\vec{n}}$ as follows. First, eliminate multipliers M_1 from all recurrences of f. Every resulting recurrence $Pf + e = 0$ implies $\sum_{n_1=0}^{n_2} P f_{\vec{n}} + \sum_{n_1=0}^{n_2} e_{\vec{n}} = 0$. Here, P is an operator polynomial that does not contain M_1, and the excess terms e do not contain f. Next, each of these recurrences is rewritten into the form $P' \sum_{n_1=0}^{n_2} f_{\vec{n}} + E_0 + E_{n_2} + \sum_{n_1=0}^{n_2} e_{\vec{n}} = 0$ where P' is an operator polynomial and the E_m's are part of excess terms built by applying some operator polynomials and the substitution $\{n_1 \mapsto m\}$ to f. This is achieved by commuting $\sum_{n_1=0}^{n_2}$ with indeterminates other than S_1 and S_2. These two indeterminates are instead handled by

$$\sum_{n_1=0}^{n_2} S_1 g_{\vec{n}} = \sum_{n_1=0}^{n_2} g_{\vec{n}} + \{n_1 \mapsto n_2 + 1\} g_{\vec{n}} - \{n_1 \mapsto 0\} g_{\vec{n}}$$
$$\sum_{n_1=0}^{n_2} S_2 g_{\vec{n}} = S_2 \sum_{n_1=0}^{n_2} g_{\vec{n}} - S_2 \{n_1 \mapsto n_2\} g_{\vec{n}}$$

Example 16. Let us continue the proof of $\sum_{n_1=0}^{n_2} \binom{n_1}{n_2-n_1} = F_{n_2+1}$ from Example 13. There we found for the summand $f_{n_1,n_2} = \binom{n_1}{n_2-n_1}$ a recurrence $(S_1 S_2^2 - S_2 - 1)f = 0$. It is actually the only recurrence after eliminating M_1 as a first step of propagation to summation. Next, we set S_1 to 1 using a telescoping identity:

$$\sum_{n_1=0}^{n_2} S_1 S_2^2 f = \sum_{n_1=0}^{n_2} S_2^2 f + \{n_1 \mapsto n_2\} S_1 S_2^2 f - \{n_1 \mapsto 0\} S_2^2 f$$

Then we push the remaining shifts S_2 leftwards:

$$\begin{aligned}\textstyle\sum_{n_1=0}^{n_2}\left(S_2^2-S_2\right)f &= S_2\textstyle\sum_{n_1=0}^{n_2}\left(S_2-1\right)f - S_2\left\{n_1\mapsto n_2\right\}\left(S_2-1\right)f\\ &= \left(S_2^2-S_2\right)\textstyle\sum_{n_1=0}^{n_2}f - S_2^2\left\{n_1\mapsto n_2\right\}f - S_2\left\{n_1\mapsto n_2\right\}\left(S_2-1\right)f\end{aligned}$$

Hence, in total we have

$$\begin{array}{lllll}\sum_{n_1=0}^{n_2}\left(S_1S_2^2-S_2-1\right)f-\left(S_2^2-S_2-1\right)\sum_{n_1=0}^{n_2}f & & & & \\ =\left\{n_1\mapsto n_2\right\}S_1S_2^2f & -\left\{n_1\mapsto 0\right\}S_2^2f & -S_2^2\left\{n_1\mapsto n_2\right\}f & -S_2\left\{n_1\mapsto n_2\right\}\left(S_2-1\right)f & \\ =\binom{n_2+1}{1} & -\binom{0}{n_2+2} & -\binom{n_2+2}{0} & -\binom{n_2+1}{1}+\binom{n_2+1}{0} & \\ =\cancel{\binom{n_2+1}{1}} & -0 & -1 & -\cancel{\binom{n_2+1}{1}}+1 & =0\end{array}$$

Since $(S_1S_2^2-S_2-1)f=0$, we have $\left(S_2^2-S_2-1\right)\sum_{n_1=0}^{n_2}f=0$. Now this is the same recurrence that F_{n_2+1} satisfies and hence the final propagation to difference gives $\left(S_2^2-S_2-1\right)\left(\sum_{n_1=0}^{n_2}f-F_{n_2+1}\right)=0$. This proves an induction step of size 2 and leaves two base cases that can be discharged by a theorem prover.

Iteration on Excess Terms. Let g be the term from the negated goal to be proved to be 0. After propagating along the structure of g, we end up with recurrences of the form $Pg=e$ where P is an operator polynomial and the excess terms e do not contain g. In the holonomic case, e will be syntactically 0. We have also observed that e is often 0 in the nonholonomic case as well. But if e is not syntactically 0, then $Pg=e$ cannot immediately be used for a proof by induction. A solution is to iterate a full series of propagations with e in place of g to find $P_2e=e_2$ and conclude $P_2Pg=P_2e=e_2$, then repeat as long as necessary. This process will always terminate, although it might fail to find recurrences.

We will impose an order on the sequence terms to accomplish three things. First, we get a proper definition of which terms in a recurrence are excess. Second, well-foundedness of the order will guarantee termination of the iteration of full propagations to excess terms. Third, the iterations can be interleaved with basic normalizations such as $\{n_1\mapsto 2n_1\}\,M_1\,\{n_1\mapsto 3n_1{+}1\}\,f \;\rightarrow\; 2M_1\,\{n_1\mapsto 6n_1{+}2\}\,f$.

Definition 17. The *spine* of a sequence term f without addition, denoted by spine f, is the sequence term obtained intuitively by erasing operator polynomials from f. Precisely, this means fully reducing f by the rewrite rules $at\rightarrow t$, $M_jt\rightarrow t$, $\{\vec{n}\mapsto b\vec{n}+\vec{c}\}t\rightarrow\{\vec{n}\mapsto b\vec{n}\}t$, and $\sum_j^{\vec{c}\bullet\vec{n}+a}t\rightarrow\sum_j^{\vec{c}\bullet\vec{n}}t$ where a, $\vec{c}$, and the matrix b are all numeric.

Shift indeterminates mix with other substitutions, which explains the last two rules. For example, spine $\{n_1\mapsto 2n_1\}\,M_1\,\{n_1\mapsto 3n_1+1\}\,f=\{n_1\mapsto 2n_1\}$ $\{n_1\mapsto 3n_1\}\,f$. If we have a sequence term $c_1g^1+\cdots+c_kg^k$ with addition, it contains multiple spines, one for each g^j. The significance of spines is that when

we derive a more complex consequence from a recurrence (during elimination by applying an operator polynomial to it), its spines do not become more complex.

We can easily describe how each propagation step changes the spines of the involved sequence terms. Propagation to the addition $f+g$ produces only spines e_f and e_g in the resulting recurrences, where e_f denotes a spine of a term from a recurrence of f and analogously for e_g. Moreover, propagation to the substitution σf produces σe_f, propagation to the product fg produces $(\text{spine}\, f)e_g$ and $e_f(\text{spine}\, g)$, and propagation to the summation $\sum_{n_1=0}^{n_2} f$ produces $\{n_1 \mapsto 0\}$ $(\text{spine}\, f)$, $\{n_1 \mapsto n_2\}\, (\text{spine}\, f)$, and $\sum_{n_1=0}^{n_2} e_f$, where e_f and e_g are as above.

We want propagations to preserve the invariant that excess terms are small. Given how spines change under propagation, a term order on spines offers a way to define smallness. We choose an order that also orients simplifications.

Definition 18. Fix a Knuth–Bendix order with argument coefficients [12] with exactly three weights $W\sum_{n=0}^{a} > W\,(\cdot) > 3W\sigma > 0$ and all argument coefficients set to 2. Moreover, projection sequence terms corresponding to M_j's (Remark 4) must have equal weights, and substitutions with fewer bindings must have lower precedence. The *excess (partial) order* on addition-free sequence terms is obtained by comparing the spines of terms using this fixed order. *Excess terms* of a recurrence are all its nonmaximal sequence terms w.r.t. the excess order.

The weights for the excess order are arranged to be compatible with normalization, which pushes substitutions to the leaf nodes of the term tree and pulls summations towards the root. The resulting normal form is simply the typical way of writing terms without explicit substitutions. It is also the normal form of the rewrite system consisting of the applicable associativity and/or commutativity rules of $\cdot$ as well as the following rules:

$$
\begin{aligned}
s \cdot \textstyle\sum_j^a t &\to \textstyle\sum_j^a st & 1t, t1, \{\}\, t &\to t \\
\left(\textstyle\sum_j^a s\right) \cdot t &\to \textstyle\sum_j^a st & (\sigma \cup \{n_j \mapsto a\})\, u &\to \sigma u \\
\sigma \textstyle\sum_j^a t &\to \textstyle\sum_j^{\sigma a} \sigma t & (\sigma \cup \{n_j \mapsto a\})\, M_j &\to a \\
\textstyle\sum_j^c t &\to \{n_j \mapsto 0\}\, t + \cdots + \{n_j \mapsto c\}\, t & \sigma\,(ts) &\to \sigma t \cdot \sigma s \\
\textstyle\sum_j^{-c} t &\to -\{n_j \mapsto -1\}\, t - \cdots - \{n_j \mapsto 1-c\}\, t & \sigma\sigma' t &\to (\sigma \circ \sigma')\, t
\end{aligned}
$$

where s, t, u are sequence terms, u does not contain the variable n_j, a is an affine variable sum, $M_j = \vec{n} \mapsto n_j$ is a projection sequence term, σ, σ' are affine substitutions, and the numeral c is nonnegative.

These rules produce additions, which must be interpreted as follows. For any rule above of the general form $t_0 \to c_1 t_1 + \cdots + c_k t_k$, the actual rewrite on the level of entire recurrences is $f\,[t_0]+R = 0 \;\to\; c_1 f\,[t_1]+\cdots+c_k f\,[t_k]+R = 0$ where c_j are numerals, the sequence terms $f\,[t_j]$ are equal except for the distinguished subterm t_j, and R is the sum of the remaining terms in the recurrence.

To conclude termination, it suffices to prove that t_0 dominates each of $t_1, \ldots,$ t_k individually. The proof is in our technical report. It makes apparent our choices of weights and argument coefficients for the transfinite Knuth–Bendix order.

5 Induction

After propagation, we consider all recurrences $Pg = 0$ of the goal sequence term g to be proved to be 0. In exceptionally fortunate cases, the operator polynomial P is ± 1 and we are unconditionally done because, for any group, the multiplication-by-± 1 map is invertible. This happens when the objective is to prove a recurrence that this method derives as a substep anyway. Otherwise, we apply induction and leave as conditions the base cases as well as invertibility of the multiplication maps associated with the leading monomials' coefficients.

A common case is that variables range over natural numbers and we have a final recurrence with leading shift $S_1^{b_1} \cdots S_d^{b_d}$ w.r.t. any monomial order. Then the values $\bigcup_{j=1}^{d} \{\vec{n} \in \mathbb{N}^d \mid n_j < b_j\}$ suffice for the base cases, as a union of stacked hyperplanes that is infinite unless $d \leq 1$, but it corresponds to only $\sum_{j=1}^{d} b_j$ one-variable substitutions $\{n_j \mapsto a\}$ for $1 \leq j \leq d$ and $0 \leq a < b_j$. If our eager generalization produced variables that do not participate in their induction (i.e., their b_j's are 0), they are replaced back to their original values.

If there is more than one applicable final recurrence, we take the intersection of their base value sets w.r.t. the same monomial order. To see that it works, consider any point outside the intersection. It is a nonbase point w.r.t. some final recurrence and hence the induction step can be taken by the recurrence.

To represent the intersection as substitutions, we distribute it over the hyperplane stack unions. This results in a union of hyperline stacks of the form $N(J, \vec{b}) := \{\vec{n} \in \mathbb{N}^d \mid n_j < b_j \text{ for all } j \in J\}$ where $J \subseteq \{1, \ldots, d\}$ and $\vec{b}$ vary. One such stack is represented by $\prod_{j \in J} b_j$ substitutions $\{n_j \mapsto a_j \mid j \in J\}$ where the a_j's are chosen arbitrarily such that $0 \leq a_j < b_j$. Unfortunately, distribution duplicates some base cases. To compensate, if $I \subseteq J$ and $\vec{b} \geq \vec{c}$ pointwise, then $N(I, \vec{b}) \supseteq N(J, \vec{c})$, so that $N(J, \vec{c})$ can be removed in favor of $N(I, \vec{b})$.

If a variable $n \in \mathbb{Z}$ is unbounded, we perform two inductions on the rays: $0 \leq n$ and $n < b$ if b base cases are needed. The backward induction on $n < b$ can be transformed into an induction on $\mathbb{N}$ by the change of variables $n \mapsto b - 1 - n$.

6 Examples

Our procedure can prove the induction step of holonomic sequence formulas such as Example 13, the binomial formula:

$$(a+b)^h = \sum_{n=0}^{h} \binom{h}{n} a^n b^{h-n} \qquad \binom{a+b}{h} = \sum_{n=0}^{h} \binom{a}{n} \binom{b}{h-n}$$

Heterogeneous recurrences, which are beyond the holonomic fragment, enable proving elementary general sequence formulas such as Example 11 and the following:

$$\sum_{n=0}^{h} f_{h-n} = \sum_{n=0}^{h} f_n \qquad \sum_{h=0}^{k} \sum_{n=0}^{h} f_{h,n} = \sum_{n=0}^{k} \sum_{h=n}^{k} f_{h,n}$$

If we ignore the holonomic base case requirements, we can for example prove the induction steps of Abel's binomial formula and of some Stirling number

identities:

$$(a+b)^h = \sum_{n=0}^{h} \binom{h}{n} a\,(a-n)^{n-1}\,(b+n)^{h-n} \qquad h^k/h! = \sum_{n=0}^{h} \left\{ {k \atop n} \right\} / (h-n)!$$

Here, the Stirling numbers of the second kind $\left\{ {k \atop n} \right\}$ are one of many special non-holonomic sequences that frequently arise in combinatorics. They count the number of partitions of a k-element set into n subsets.

As further demonstration, we apply our procedure to the last equation. For convenience, we will use the name of a variable also to denote its multiplier operator. Moreover, we will use the uppercase version of the name of a variable to denote its shift operator. The defining recurrence of the Stirling numbers then reads $(KN - (n+1)\,N - 1) \left\{ {k \atop n} \right\} = 0$ for $k, n \geq 0$, where K and N denote the shift operators for the variables k and n, the first n denotes the multiplier for the variable n, and the second n is the variable itself. This recurrence is complemented by the initial values $\left\{ {0 \atop 0} \right\} = 1$ and $\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 0$ if $n \neq 0$.

Starting from the right, the inverse $m!^{-1}$ of the factorial satisfies the recurrence $(mM + M - 1)\,m!^{-1} = 0$ that holds for all $m \in \mathbb{Z}$ by extension. This must be found in the initialization step because there is no propagation to division. Propagation to the substitution $\{m \mapsto h - n\}$ then gives the following recurrences, factored for clarity:

$$((h-n+1)\,H - 1)\,(h-n)!^{-1} = 0 \qquad (N - h + n)\,(h-n)!^{-1} = 0$$

To propagate to product, we consider $\left\{ {k_1 \atop n_1} \right\}$ and $(h_2 - n_2)!^{-1}$ with variables renamed apart. We must propagate to the substitution $\{n_j \mapsto n,\ h_j \mapsto h,\ k_j \mapsto k \mid j \in \{1,2\}\}$ the recurrences of $\left\{ {k_1 \atop n_1} \right\} (h_2 - n_2)!^{-1}$ given by the following five operator polynomials:

$$\begin{array}{rl} K_1 N_1 - (n_1+1)\,N_1 - 1 \ \text{ and } \ H_1 - 1 & \text{from } \left\{ {k_1 \atop n_1} \right\} \\ (h_2 - n_2 + 1)\,H_2 - 1,\ N_2 - h_2 + n_2, \ \text{ and } \ K_2 - 1 & \text{from } (h_2 - n_2)!^{-1} \end{array}$$

We added here the trivial recurrences given by $H_1 - 1$ and $K_2 - 1$ implied by the independence from h_1 and k_2. Among the defining recurrences (4) of the compound shift indeterminates N, H, K, the recurrence $H_1 H_2 - H$ simplifies to $H_2 - H$ by $H_1 - 1$ and $K_1 K_2 - K$ to $K_1 - K$ by $K_2 - 1$. (In other words, the factorwise renaming of already disjoint variables h and k amounts to renaming in the entire product.) The third compound shift recurrence, $N_1 N_2 - N$, simplifies to $(h_2 - n_2)\,N_1 - N$ by $N_2 - h_2 + n_2$. The part of the Gröbner basis with only compound shifts is then straightforwardly finished with the result $\{KN - (n_1+1)\,N - h_2 + n_2,\ (h_2 - n_2 + 1)\,H - 1\}$. Hence this propagation step yields

$$(KN - (n+1)\,N - h + n)\,\frac{\left\{ {k \atop n} \right\}}{(h-n)!} = 0 \quad ((h-n+1)\,H - 1)\,\frac{\left\{ {k \atop n} \right\}}{(h-n)!} = 0$$

To sum over n, we first eliminate n from the previous two recurrences and conclude $(H\,(K-h) + (N-1)\,(KH - (h+1)\,H + 1))\,(\left\{ {k \atop n} \right\} / (h-n)!) = 0$. The

sum has natural boundaries, meaning that the summand vanishes outside them. This guarantees that there will be no excess terms, which we also tediously discover when pulling out the indeterminates:

$$\sum_{n=0}^{h}(N-1)\overbrace{(KH-(h+1)H+1)}^{P}\frac{\left\{{k \atop n}\right\}}{(h-n)!} = P\left(\frac{\left\{{k \atop h+1}\right\}}{(-1)!}-\frac{\left\{{k \atop 0}\right\}}{h!}\right)$$
$$= -\left\{{k+1 \atop 0}\right\}/(h+1)! + \left\{{k \atop 0}\right\}((h+1)H-1)h!^{-1} = 0$$
$$\sum_{n=0}^{h}H(K-h)\frac{\left\{{k \atop n}\right\}}{(h-n)!} - H(K-h)\sum_{n=0}^{h}\frac{\left\{{k \atop n}\right\}}{(h-n)!} = -(K-h)\frac{\left\{{k \atop h+1}\right\}}{(-1)!} = 0$$

Here, by the recurrence of the inverse of the factorial, we get $(-1)!^{-1} = 0$. So we obtain a recurrence $H(K-h)\sum_{n=0}^{h}\left\{{k \atop n}\right\}/(h-n)! = 0$ for the left-hand side of our goal. For the right-hand side, we unproblematically obtain $(K-h)\big(h^k/h!\big) = 0$. Hence $H(K-h)$ zeros out the difference $h^k/h! - \sum_{n=0}^{h}\left\{{k \atop n}\right\}/(h-n)!$. The largest shift HK of the operator $H(K-h)$ determines that the two sets of base cases $h = 0$ and $k = 0$ are sufficient for induction.

7 Related Work

Holonomic sequences [20] are closely related to our work. Unlike our approach, which allows infinitely many base cases as long as they are finitely representable (Sect. 5), they are limited to a finite number of base cases. Relaxing this limitation yields approximately the homogeneous version of our propagation procedure (i.e., without excess terms), whose theory Chyzak, Kauers, and Salvy laid out [6]. Heterogeneity amounts to module Gröbner bases [5,8,13]. Its integration into propagations makes elementary identities about general sequences automatically provable, which may be of interest for general-purpose theorem provers.

In practice, hypergeometric sums are common holonomic sequences that have much faster algorithms available. Gosper's indefinite summation [9] can be applied to compute Wilf–Zeilberger pairs [19], which offer compact proof certificates for definite sum identities. These fast methods admit generalizations to the full holonomic setting. See Koutschan's thesis [11] for an overview.

Finding a closed form instead of only checking it for a summation is a different but related task. A common approach is to perform a recurrence solving phase after recurrence computation, as in the Mathematica package Sigma [1,15].

8 Conclusion

We presented a procedure for proving equations involving summations within an automatic higher-order theorem prover. The procedure is inspired by holonomic sequences and partly generalizes them. It expresses the problem as recurrences and derives new recurrences from existing ones. In case of success, it shows the induction step of a proof by induction, leaving the base cases to the prover.

As future work, we want to continue implementing the procedure in Zipperposition [17]. We hope that the subsequent practical experiments help us to settle how side conditions of initial recurrences ought to be handled.

Acknowledgment. We thank Pascal Fontaine for his ideas on how to integrate our procedure into SMT and tableaux. We also thank Anne Baanen, Pascal Fontaine, Mark Summerfield, and the anonymous reviewers for suggesting improvements.

Nummelin and Blanchette's research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward). Dahmen's research has received funding from the NWO under the Vidi program (project No. 639.032.613, New Diophantine Directions).

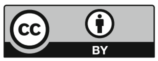
References

1. Abramov, S.A., Bronstein, M., Petkovsek, M., Schneider, C.: On rational and hypergeometric solutions of linear ordinary difference equations in ΠΣ*-field extensions. J. Symb. Comput. **107**, 23–66 (2021)
2. Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 35–54. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_3
3. 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
4. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formaliz. Reason. **9**(1), 101–148 (2016)
5. Bueso, J., Gómez-Torrecillas, J., Verschoren, A.: Gröbner bases for modules. In: Bueso, J., Gómez-Torrecillas, J., Verschoren, A. (eds.) Algorithmic Methods in Non-Commutative Algebra: Applications to Quantum Groups, pp. 169–202. Springer, Dordrecht (2003). https://doi.org/10.1007/978-94-017-0285-0_5
6. Chyzak, F., Kauers, M., Salvy, B.: A non-holonomic systems approach to special function identities. In: Johnson, J.R., Park, H., Kaltofen, E. (eds.) Symbolic and Algebraic Computation, International Symposium, ISSAC 2009, Seoul, Republic of Korea, 29–31 July 2009, pp. 111–118. ACM (2009)
7. Cox, D.A., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Undergraduate Texts in Mathematics, 4th edn. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16721-3
8. Fajardo, W., Gallego, C., Lezama, O., Reyes, A., Suárez, H., Venegas, H.: Gröbner bases of modules. In: Gallego, C., Lezama, O., Reyes, A., Suárez, H., Venegas, H. (eds.) Skew PBW Extensions. AA, vol. 28, pp. 261–286. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53378-6_14
9. Gosper, R.W.: Decision procedure for indefinite hypergeometric summation. Proc. Natl. Acad. Sci. USA **75**(1), 40–42 (1978)
10. Kandri-Rody, A., Weispfenning, V.: Non-commutative Gröbner bases in algebras of solvable type. J. Symb. Comput. **9**(1), 1–26 (1990)
11. Koutschan, C.: Advanced applications of the holonomic systems approach. ACM Comm. Comput. Algebra **43**(3/4), 119 (2009)
12. Ludwig, M., Waldmann, U.: An extension of the Knuth-Bendix ordering with LPO-Like properties. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 348–362. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75560-9_26

13. Maletzky, A., Immler, F.: Gröbner bases of modules and Faugère's f_4 algorithm in Isabelle/HOL. CoRR abs/1805.00304 (2018)
14. Nummelin, V., Blanchette, J., Dahmen, S.R.: Automated deduction with recurrence-driven summations (technical report). Technical report (2023). https://lean-forward.github.io/pubs/sums_report.pdf
15. Schneider, C.: Symbolic summation assists combinatorics. Sem. Lothar. Combin. **56**, 1–36 (2007)
16. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. J. Autom. Reason. **65**(6), 775–807 (2021)
17. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. J. Autom. Reason. **66**(4), 541–564 (2022)
18. Vukmirović, P., Blanchette, J., Schulz, S.: Extending a high-performance prover to higher-order logic. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13994, pp. 111–129. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_10
19. Wilf, H.S., Zeilberger, D.: Rational functions certify combinatorial identities. J. Am. Math. Soc. **3**(1), 147–158 (1990)
20. Zeilberger, D.: A holonomic systems approach to special functions identities. J. Comput. Appl. Math. **32**(3), 321–368 (1990)

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.



Formal Verification of Bit-Vector Invertibility Conditions in Coq

Burak Ekici[1], Arjun Viswanathan[2], Yoni Zohar[3](✉), Cesare Tinelli[2], and Clark Barrett[4]

[1] Muğla Sıtkı Koçman University, Muğla, Turkey
[2] The University of Iowa, Iowa City, USA
[3] Bar-Ilan University, Ramat Gan, Israel
yoni.zohar@biu.ac.il
[4] Stanford University, Stanford, USA

Abstract. We prove the correctness of invertibility conditions for the theory of fixed-width bit-vectors—used to solve quantified bit-vector formulas in the Satisfiability Modulo Theories (SMT) solver cvc5— in the Coq proof assistant. Previous work proved many of these in a completely automatic fashion for arbitrary bit-width; however, some were only proved for bit-widths up to 65, even though they are being used to solve formulas over larger bit-widths. In this paper we describe the process of proving a representative subset of these invertibility conditions in Coq. In particular, we describe the `BVList` library for bit-vectors in Coq, our extensions to it, and proofs of the invertibility conditions.

1 Introduction

Many applications in hardware and software verification rely on bit-precise reasoning, which can be modeled using the SMT-LIB 2 theory of fixed-width bit-vectors [3]. While Satisfiability Modulo Theories (SMT) solvers are able to reason about bit-vectors of fixed width, they currently require all widths to be expressed concretely (by a numeral) in their input formulas. For this reason, they cannot be used to prove properties of bit-vector operators that are parametric in the bit-width, such as the associativity of bit-vector concatenation. Proof assistants such as Coq [25], which have direct support for dependent types, are better suited for such tasks.

Bit-vector formulas that are parametric in the bit-width arise in the verification of parametric Boolean functions and circuits (see, e.g., [13]). In our case, we are mainly interested in parametric lemmas that are relevant to internal techniques of SMT solvers for the theory of fixed-width bit-vectors. These include, for example, rewrite rules, refinement schemes, and preprocessing passes. Such techniques are developed a priori for every possible bit-width. Meta-reasoning about the correctness of such solvers then requires bit-width independent reasoning.

In this paper, we focus on parametric lemmas that originate from a quantifier-instantiation technique implemented in the SMT solver cvc5 [2]. This technique

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 41–59, 2023.
https://doi.org/10.1007/978-3-031-43369-6_3

is based on *invertibility conditions* [15]. For a trivial case of an invertibility condition, consider the equation $x + s = t$. where x, s and t are variables of the same bit-vector sort. In the terminology of Niemetz et al. [15], this equation is "invertible for x." A general inverse, or "solution," is given by the term $t-s$. Since there is always such an inverse, the invertibility condition for $x + s = t$ is simply the universally true formula $\top$. The formula stating this fact, referred to here as an *invertibility equivalence*, is $\top \Leftrightarrow \exists x.\ x + s = t$, which is valid in the theory of fixed-width bit-vectors, for any bit-width. In contrast, the equation $x \cdot s = t$ is not always invertible for x. A necessary and sufficient condition for invertibility in this case was found in [15] to be $(-s \mid s)\ \&\ t = t$. So, the invertibility equivalence $(-s \mid s)\ \&\ t = t \Leftrightarrow \exists x.\ x \cdot s = t$ is valid for any bit-width. Notice that the invertibility condition does not contain x. Hence, invertibility conditions can be seen as a technique for quantifier elimination.

In [15], a total of 160 invertibility conditions were provided. However, they were verified only for bit-widths up to 65, due to the reasoning limitations of SMT solvers mentioned earlier. Recent work [16,17] addresses this challenge by translating the invertibility equivalences to the combined theory of non-linear integer arithmetic and uninterpreted functions. This approach was partially successful, but failed to verify over a quarter of the equivalences.

We verify invertibility equivalences proposed in [15] by proving them interactively in Coq. From a representative subset of the invertibility equivalences, we prove 19 equivalences, 12 of which were not proven in [16,17]. For the remaining 7, that were already proved there, our Coq proofs provide more confidence. Our results offer evidence that proof assistants can support automated theorem provers in meta-verification tasks. To facilitate the verification of invertibility equivalences, we use a rich Coq library for bit-vectors, which is a part of the SMTCoq project [10]. This Coq library models the theory of fixed-width bit-vectors adopted by the SMT-LIB 2 standard [3]. For this work, we extended the library with the arithmetic right-shift operation and the unsigned weak less-than

Table 1. The signatures Σ_1 and Σ_0 with SMT-LIB 2 syntax. Σ_1 consists of the operators in the entire table. Σ_0 consists of the operators in the upper part.

Symbol	SMT-LIB Syntax	Sort
$=, \neq$	=, distinct	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$<_u, >_u, \leq_u, \geq_u$	bvult, bvugt, bvule, bvuge	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$\sim, -$	bvnot, bvneg	$\sigma_{[n]} \to \sigma_{[n]}$
$\&, \mid, \ll, \gg, \gg_a$	bvand, bvor, bvshl, bvlshr, bvashr	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$+$	bvadd	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$<_s, >_s, \leq_s, \geq_s$	bvslt, bvsgt, bvsle, bvsge	$\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$
$\cdot$, mod, $\div$	bvmul, bvurem, bvudiv	$\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$
$\circ$	concat	$\sigma_{[n]} \times \sigma_{[m]} \to \sigma_{[n+m]}$
$[u : l]$	extract	$\sigma_{[n]} \to \sigma_{[u-l+1]}$

and greater-than predicates. To summarize, the contributions of this paper are as follows: (*i*) a description of the SMTCoq bit-vector library; (*ii*) extensions to the signature and proofs of the library; and (*iii*) formal proofs in Coq of invertibility equivalences. These contributions, while important in their own right, have the potential to go beyond the verification of invertibility equivalences. For (*i*) and (*ii*), we envision that the library, as well as its extension, will be useful for the formalization of other bit-precise reasoning mechanisms, especially related to SMT, such as rewriting rules, lemma schemas, interactive verification, and more. For (*iii*), invertibility conditions are primarily used for quantifier instantiation (see, e.g., [15]). We hope that the increased confidence in their correctness will encourage their usage in other contexts and in more solvers. Further, the formal proofs can serve as guiding examples for other proofs related to bit-precise reasoning.

The remainder of this paper is organized as follows. After technical preliminaries in Sect. 2, we formalize invertibility conditions in Sect. 3 and discuss previous attempts at verifying them. In Sect. 4, we describe the Coq library and our extensions to it. In Sect. 5, we discuss our Coq proofs. We conclude in Sect. 6 with directions for future work. A preliminary version of this work was presented as an extended abstract in the proceedings of the PxTP 2019 workshop [11]. The current version is more detailed and complete. In particular, the one Coq proof that was missing in [11] is now completed.

2 Preliminaries

2.1 Theory of Bit-Vectors

We assume the usual terminology of many-sorted first-order logic with equality (see, e.g., [12]). We denote equality by $=$, and use $x \neq y$ as an abbreviation for $\neg(x = y)$. The signature Σ_{BV} of the SMT-LIB 2 theory of fixed-width bit-vectors defines a unique sort for each positive integer n, which we denote by $\sigma_{[n]}$. For every positive integer n and bit-vector of width n, the signature contains a constant symbol of sort $\sigma_{[n]}$, representing that bit-vector, which we denote as a binary string of length n. The function and predicate symbols of Σ_{BV} are as described in the SMT-LIB 2 standard. Formulas of Σ_{BV} are built from variables, bit-vector constants, and the function and predicate symbols of Σ_{BV}, along with the usual logical connectives and quantifiers. We write $\psi[x_1, \ldots, x_n]$ to represent a formula whose free variables are from the set $\{x_1, \ldots, x_n\}$.

The semantics of Σ_{BV}-formulas is given by interpretations where the domain of $\sigma_{[n]}$ is the set of bit-vectors of width n, and the function and predicate symbols are interpreted as specified by the SMT-LIB 2 standard. A Σ_{BV}-formula is *valid* in the theory of fixed-width bit-vectors if it is satisfied by every such interpretation.

Table 1 contains the operators from Σ_{BV} for which invertibility conditions were defined in [15]. We define Σ_1 to be the signature that contains only these symbols. Σ_0 is the sub-signature obtained by only taking the operators from the

upper part of the table. We use the (overloaded) constant 0 to represent the bit-vectors composed of all 0-bits.

2.2 Coq

The Coq proof assistant is based on the calculus of inductive constructions (CIC) [20]. It implements properties as types, and proofs as terms, reducing proof-checking to type-checking. Coq has a rich type system, that allows for highly expressive propositions to be stated and proved in this manner. One particular feature of interest is that of *dependent types* — types that can depend on values — through which one can express correctness properties within types. We refer to non-dependent types as *simple types*.

The Coq module system — in addition to allowing for principled separations of large developments — allows the abstraction of complex types along with operations over them as *modules*. A *module signature* or *module type* acts as an interface to a module, specifying the type it encapsulates along with the signatures of the associated operators. A *functor* is a module-to-module function.

3 Invertibility Conditions and Their Verification

In [15], a technique to solve quantified bit-vector formulas is presented, which is based on *invertibility conditions*.

Definition 1. *An invertibility condition for a variable x in a Σ_{BV}-literal $\ell[x, s, t]$ is a formula $IC[s, t]$ such that $\forall s.\forall t.\ IC[s, t] \Leftrightarrow \exists x.\ \ell[x, s, t]$ is valid in the theory of fixed-width bit-vectors.*

Example 1. The invertibility condition for x in $x \,\&\, s = t$ is $t \,\&\, s = t$. □

In [15], invertibility conditions are defined for a representative set of literals ℓ over the bit-vector operators of Σ_1, having a single occurrence of x. The soundness of the technique proposed in that work relies on the correctness of the invertibility conditions. Every literal $\ell[x, s, t]$ and its corresponding invertibility condition $IC[s, t]$ induce an *invertibility equivalence*.

Definition 2. *The* invertibility equivalence *associated with the literal $\ell[x, s, t]$ and its invertibility condition $IC[s, t]$ is the formula*

$$IC[s, t] \Leftrightarrow \exists x.\ \ell[x, s, t] \tag{1}$$

The correctness of invertibility equivalences should be verified for all possible sorts for the variables x, s, t for which the condition is well sorted. Concretely, one needs to prove the validity of the following formula:

$$\forall n : \mathbb{N}.\ n > 0 \Rightarrow \forall s : \sigma_{[n]}.\forall t : \sigma_{[n]}.\ IC[s, t] \Leftrightarrow \exists x : \sigma_{[n]}.\ \ell[x, s, t] \tag{2}$$

This was done in [15], but only for concrete values of n from 1 to 65, using solvers for the theory of fixed-width bit-vectors. In contrast, Eq. (2) cannot even

be expressed in this theory. To overcome this limitation, later work suggested a translation from bit-vector formulas over *parametric* bit-widths to the theory of non-linear integer arithmetic with uninterpreted functions [16,17]. Thanks to this translation, the authors were able to verify the correctness of 110 out of 160 invertibility equivalences. For the remaining 50 equivalences, it then seems appropriate to use a proof-assistant, as this allows for more intervention by the user who can provide crucial intermediate steps. Even for the 110 invertibility equivalences that were proved, the level of confidence achieved by proving them in a proof assistant would be greater than an automatic verification by an SMT solver due to the smaller trusted code-base of proof assistants in relation to those of automatic theorem provers such as SMT solvers.

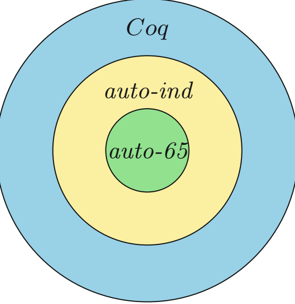


Fig. 1. The level of confidence achieved by the different approaches.

Figure 1 depicts the level of confidence achieved by the various approaches to verify invertibility equivalences. The smallest circle, labelled *auto-65*, represents the approach taken by [15], where invertibility equivalences were verified automatically up to 65 bits. While a step in the right direction, this approach is insufficient, because invertibility conditions are used for arbitrary bit-widths. The next circle, labeled *auto-ind*, depicts the approach of [17], which addresses the restrictions of auto-65 by providing bit-width independent proofs of the invertibility equivalences. However, both auto-65 and auto-ind provide proofs by SMT solvers, which are less trusted than ITPs. The largest circle (*Coq*) corresponds to work presented in the current paper which, while addressing the limitations of auto-65 via bit-width independent proofs, also provides stronger verification guarantees by proving the equivalences in an interactive theorem prover. Moreover, with this approach, we were able to prove equivalences that couldn't be fully verified (for arbitrary bit-widths) by either auto-65 or auto-ind.

4 The `BVList` Library

In this section, we describe the Coq library we use and the extensions we developed with the goal of formalizing and proving invertibility equivalences. Vari-

ous formalizations of bit-vectors in Coq exist. The internal Coq library of bit-vectors [9] is one, but it has only definitions and no lemmas. The Bedrock Bit Vectors Library [6] treats bit-vectors as words (machine integers). The SSRBit Library [5] represents bit-vectors as finite bit-sets in Coq and extracts them to OCaml machine integers. Our library is more suited to the SMT-LIB 2 bit-vectors, and includes operators that are not fully covered by any of the previously mentioned libraries. More recently, Shi et al. [22] developed a library called CoqQFBV that presents a bit-vector type as a sequence of Booleans, defines operators over it, and proves the correctness of these operations with respect to a (machine integer) semantics. [22] uses this library to define a bit-blasting algorithm in Coq, that is extracted into an OCaml program to perform certified bit-blasting. Since CoqQFBV covers the entire SMT-LIB 2 bit-vector signature, it would be a good alternative to ours in formalizing and proving invertibility conditions. Our library offers a rich set of lemmas over bit-vector operations that makes it suitable for proofs of invertibility conditions and other bit-vector properties. Bit-vectors have also been formalized in other proof assistants. Within the Isabelle/HOL framework, one can utilize the library developed by Beeren et al. [4] to align with SMT-LIB 2 bit-vector operations. Furthermore, Harrison [1] presents a formalization of finite-dimensional Euclidean space within HOL light, accompanied by an implementation of vectors.

4.1 `BVList` Without Extensions

`BVList` was developed for SMTCoq [10], a Coq plugin that enables Coq to dispatch proofs to external proof-producing solvers. While the library was only briefly mentioned in [10], here we provide more details.

The library adopts the little-endian notation for bit-vectors, following the internal representation of bit-vectors in SMT solvers such as cvc5, and corresponding to lists in Coq. This makes arithmetic operations easier to perform since the least significant bit of a bit-vector is the head of the Boolean list that represents it.

Another choice is how to formalize the bit-vector type. A dependently-typed definition is natural, since then the type of a bit-vector is parameterized by its length. However, such a representation leads to some difficulties in proofs. Dependent pattern-matching or case-analysis with dependent types is cumbersome and unduly complex (see, e.g., [23]), because of the complications brought by unification in Coq (which is inherently undecidable [24]). A simply-typed definition, on the other hand, does not provide such obstacles for proofs, but is less natural, as the length becomes external to the type. The `BVList` library defines for convenience both the dependently and the simply typed version of bit-vectors. It uses the Coq module system to separate them, and a functor that connects them, avoiding redundancy. The relationship between the two definitions is depicted in Fig. 2.

In `BVList`, a dependently-typed bit-vector is a record parameterized by its size n and consisting of two fields: a Boolean list and a condition to ensure that the list has length n. This type, and the corresponding lemmas and properties over it, are encapsulated by the `BITVECTOR_LIST` module of type `BITVECTOR`. A

simply-typed or *raw* bit-vector representation is simply a Boolean list which, along with its associated operators and lemmas is specified by module signature `RAWBITVECTOR` and implemented in module `RAWBITVECTOR_LIST`. In other words, the interface of `BVList` offers dependently-typed bit-vectors, while the underlying operators are defined and proofs are performed using raw bit-vectors.

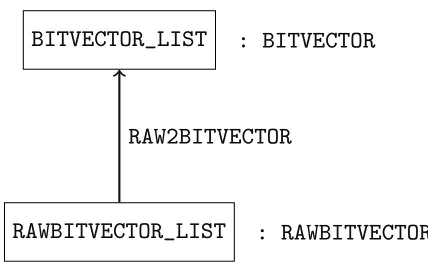


Fig. 2. Modular separation of `BVList`

A functor called `RAW2BITVECTOR` derives corresponding definitions and proofs over dependently-typed bit-vectors within the module for dependent-types, when it is applied to `RAWBITVECTOR_LIST`. The functor establishes a correspondence between the two theories so that one can first prove a bit-vector property in the context of the simply-typed theory and then map it to its corresponding dependently-typed one via the functor module. Otherwise put, users of the library can encode theorem statements more naturaly, and in a more expressive environment employing dependent types. For proofs, one can unlift them (by the functor) to the equivalent encodings with simple types, and prove them there.

4.2 Extending `BVList`

Out of the 13 bit-vector functions and 10 predicates contained in Σ_1, `BVList` had direct support for 10 functions and 6 predicates. The predicate symbols that were not directly supported were the weak inequalities $\leq_u$, $\geq_u$, $\leq_s$, $\geq_s$ and the unsupported function symbols were $\gg_a$, $\div$, and $\bmod$. We extended `BVList` with the operator $\gg_a$ and the predicates $\leq_u$ and $\geq_u$ in order to support the corresponding invertibility conditions. Additionally, we redefined $\ll$ and $\gg$ in order to simplify the proofs of invertibility conditions over them.[1]

We focused on invertibility conditions for literals of the form $x \diamond s \bowtie t$ and $s \diamond x \bowtie t$, where $\diamond$ and $\bowtie$ are respectively function and predicate symbols in Σ_0. Σ_0 was chosen as a representative set because it is both expressive enough (in the sense that other operators can be easily translated to this fragment), and

[1] Both the extended library and the proofs of invertibility equivalences can be found at https://github.com/ekiciburak/bitvector/tree/frocos23.

feasible for proofs in Coq using the library. In particular, it was chosen as one that would require the minimal amount of changes to `BVList`. As a result, such literals, as well as their invertibility conditions, contain only operators supported by `BVList` (after its extension with $\gg_a$, $\leq_u$, and $\geq_u$). Supporting the full set of operators in Σ_1, both in the library and the proofs is left for future work.

```
1  Fixpoint ule_list_big_endian (x y : list bool) :=
2    match x, y with
3     | [ ], [ ] ⇒ true
4     | [ ], _  ⇒ false
5     | _, [ ] ⇒ false
6     | xi:: x',  yi:: y' ⇒ ((eqb xi yi) && (ule_list_big_endian x' y'))
7                          ||  ((negb xi) && yi)
8    end.
9
10 Definition ule_list (x y: list bool) :=
11   (ule_list_big_endian (rev x) (rev y)).
12
13 Definition bv_ule (a b : bitvector) :=
14   if @size a =? @size b then
15     ule_list a b
16   else
17     false.
18
19 Definition bv_ule n (bv1 bv2:bitvector n) : bool := M.bv_ule bv1 bv2.
```

Fig. 3. Definitions of $\leq_u$ in Coq.

In what follows, we describe our extensions to `BVList` with weak unsigned inequalities, alternative definitions for logical shifts, and the arithmetic right shift operator.

Weak Unsigned Inequalities. We added both weak inequalities for unsigned bit-vectors, $\leq_u$ and $\geq_u$. We illustrate this extension via that of the $\leq_u$ operator (the extension of $\geq_u$ is similar). The relevant Coq definitions are provided in Fig. 3. The top three definitions (including the fixpoint) cover the simply-typed representation, and the fourth, `bv_ule` is the dependently-typed representation that invokes the definition with the same name from module `M` of type `RAWBITVECTOR`. Like most other operators, $\leq_u$ (over raw bit-vectors) is defined over a few *layers*. The function `bv_ule`, at the highest layer, ensures that comparisons are between bit-vectors of the same size and then calls `ule_list`. Since we want to compare bit-vectors starting from their most significant bits and the input lists start instead with the least significant bits, `ule_list` first reverses the two lists. Then it calls `ule_list_big_endian`, which we consider to be at the lowest layer of the definition. This function does a lexicographic comparison of the two lists, starting from the most significant bits.

To see why the addition of $\leq_u$ to the library is useful, consider, for example, the following parametric lemma, stating that ~ 0 is the largest unsigned bit-vector of its type:

$$\forall x : \sigma_{[n]}.\ x \leq_u \sim 0 \tag{3}$$

Without an operator for the weak inequality, we would write it as:

$$\forall x : \sigma_{[n]}.\ x <_u \sim 0 \lor x = \sim 0 \tag{4}$$

```
1   Definition shl_one_bit (a: list bool) :=
2     match a with
3      | [ ] ⇒ [ ]
4      | _ ⇒ false :: removelast a
5     end.
6
7   Fixpoint shl_n_bits (a: list bool) (n: nat) :=
8     match n with
9      | O ⇒ a
10     | S n' ⇒ shl_n_bits (shl_one_bit a) n'
11    end.
12
13  Definition shl_n_bits_a (a: list bool) (n: nat) :=
14    if (n <? length a)%nat then
15      mk_list_false n ++ firstn (length a -n) a
16    else
17      mk_list_false (length a).
18
19  Theorem bv_shl_eq: forall (a b : bitvector), bv_shl a b = bv_shl_a a b.
```

Fig. 4. Various definitions of $\ll$.

In such cases, since the definitions of $<_u$ and $=$ have a similar structure to that of $\leq_u$, we strip down the layers of $<_u$ and $=$ separately, whereas using $\leq_u$, we only do this once.

Left and Right Logical Shifts. We have redefined the shift operators $\ll$ and $\gg$ in `BVList`. Figure 4 shows both the original and new definitions of $\ll$. Those of $\gg$ are similar. Originally, $\ll$ was defined using the `shl_one_bit` and `shl_n_bits`. The function `shl_one_bit` shifts the bit-vector to the left by one bit and is called by `shl_n_bits` as many times as necessary. The new definition `shl_n_bits_a` uses `mk_list_false` which constructs the necessary list of 0 bits and appends (`++` in Coq) to it the bits to be shifted from the original bit-vector, which are retrieved using the `firstn` function, from the Coq standard library for lists. The `nat` type used in Fig. 4 is the Coq representation of Peano natural numbers that has `O` and `S` as its two constructors — as depicted in the cases rendered by pattern matching n (lines 9-10). The theorem at the bottom of

Fig. 4 asserts the equivalence of the two representations, allowing us to switch between them, when needed. In the extended library, `bv_shl` defines the left shift operation using `shl_n_bits` whereas `bv_shl_a` does it using `shl_n_bits_a`. This new representation was useful in proving some of the invertibility equivalences over shift operators (see, e.g., Example 4 below).

Arithmetic Right Shift. Unlike logical shifts that were already defined in `BVList` and for which we have added alternative definitions, arithmetic right shift was not defined at all. We provided two alternative definitions for it, very similar to the definitions of logical shifts — `bv_ashr` and `bv_ashr_a`. Both definitions are conditional on the sign of the bit-vector (its most-significant bit). Apart from this detail, the definitions take the same approach taken by `shl_n_bits` and `shl_n_bits_a` from Fig. 4. Operator `bv_ashr` uses the definition of an independent shift and repeats it as many number of times as necessary, and `bv_ashr_a` uses either `mk_list_false` or `mk_list_true` to append the necessary number of sign bits to the shifted bits.

5 Proving Invertibility Equivalences in Coq

In this section we provide specific details about proving invertibility equivalences in Coq. We start by outlining the general approach for proving invertibility equivalences in Sect. 5.1. Then, Sect. 5.2 presents detailed examples of such proofs. Section 5.3 summarizes the results and impact of these proofs.

5.1 General Approach

The natural representation of bit-vectors in Coq is the dependently-typed representation, and therefore the invertibility equivalences are formulated using this representation. In keeping with the modular approach described in Sect. 4, however, proofs in this representation are composed of proofs over simply-typed bit-vectors, which are easier to reason about. Most of the work is on proving an equivalence over raw bit-vectors. Then, we derive the proof of the corresponding equivalence over dependently-typed bit-vectors using a smaller, boilerplate set of tactics. Since this derivation process is mostly the same across many equivalences, these tactics are a good candidate for automation in the future.

When proving an invertibility equivalence $IC[s,t] \Leftrightarrow \exists x.\ \ell[x,s,t]$, we first split it into two sub-goals: the left-to-right and right-to-left implications. For proving the left-to-right implication, since Coq implements a constructive logic, the only way to prove an existentially quantified formula is to construct the literal witnessing it. Thus, in addition to being able to prove the equivalence, a positive side-effect of our proofs are actual inverses for x in literals of the form $\ell[x,s,t]$. In Niemetz et al. [16], these are called *conditional inverses*, as the fact that they are inverses is conditional on the correctness of the invertibility condition. There, such inverses were synthesized automatically for a subset of the literals. In each of our Coq proofs, such an inverse is found, even when the

proof is done by case-splitting. This provides a more general solution than the one in [16], which did not consider case-splitting.

Example 2. Consider the literal $s \gg_a x \geq_u t$. Its invertibility condition is $(s \geq_u {\sim}s) \vee (s \geq_u t)$. The left-to-right implication of the invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (s \geq_u {\sim}s) \vee (s \geq_u t) \Rightarrow \exists x : \sigma_{[n]}.\ s \gg_a x \geq_u t$$

Here, case splitting is done on the disjunction in the invertibility condition. When $s \geq_u {\sim}s$ is true, the inverse for x is the bit-vector constant that correspond to the length of the s, namely n; when $s \geq_u t$ is true, the inverse is 0. □

In addition to `BVList`, several proofs of invertibility equivalences benefited from CoqHammer [7], a plug-in that aims at extending the level of automation in Coq by combining machine learning and automated reasoning techniques in a similar fashion to what is done in by Sledgehammer [21] in Isabelle/HOL [18]. CoqHammer, when triggered on some Coq goal, (i) submits the goal together with potentially useful terms to external solvers/automated-provers, (ii) attempts to reconstruct returned proofs (if any) directly in the Coq tactic language Ltac [8], and (iii) outputs the set of tactics closing the goal in case of success. As we directly employ these tactics inside `BVList`, one does not need to install CoqHammer in order to build the library, although it would be beneficial for further extensions.

5.2 Detailed Examples

In this section we provide specific examples for proofs of invertibility equivalences. The first example illustrates the two-theories approach of the library.

Example 3. Consider the literal $s \gg_a x <_u t$. Its invertibility condition is $((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$. Figure 5 shows the proof of the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ (\exists x : \sigma_{[n]}.\ s \gg_a x <_u t) \Rightarrow ((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$$

In the proof, lines 8–11 transform the dependent bit-vectors from the goal and the hypotheses into simply-typed bit-vectors. Then, lines 12–14 invoke the corresponding lemma for simply-typed bit-vectors (called `InvCond.bvashr_ult2_rtl`) along with some simplifications. □

Most of the effort in this project went into proving equivalences over raw bit-vectors, as the following example illustrates.

Example 4. Consider the literal $x \ll s >_u t$. Its invertibility condition is $(t <_u {\sim}0 \ll s)$. The corresponding invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (t <_u {\sim}0 \ll s) \Leftrightarrow (\exists x : \sigma_{[n]}.\ x \ll s >_u t) \tag{5}$$

The left-to-right implication is easy to prove using ~ 0 itself as the witness of the existential proof goal and considering the symmetry between $>_u$ and $<_u$. The proof of the right-to-left implication relies on the following lemma:

$$\forall x, s : \sigma_{[n]}.\ (x \ll s) \leq_u (\sim 0 \ll s) \tag{6}$$

From the right side of the equivalence in Eq. (5), we get some skolem x for which $x \ll s >_u t$ holds. Flipping the inequality, we have that $t <_u x \ll s$; using this, and transitivity over $<_u$ and $\leq_u$, the lemma given by Eq. (6) gives us the left side of the equivalence in Eq. (5).

As mentioned in Sect. 4, we have redefined the shift operators $\ll$ and $\gg$ in the library. This was instrumental, for example, in the proof of Eq. (6).

```
1  Theorem bvashr_ult2_rtl :
2   forall (n : N), forall (s t : bitvector n),
3   (exists (x : bitvector n), (bv_ult (bv_ashr_a s x) t = true)) ->
4   ((( bv_ult s t = true) ∨ (bv_slt s (zeros n)) = false) ∧
5   (bv_eq t (zeros n)) = false).
6  Proof.
7   intros n s t H.
8   destruct H as ((x, Hx), H).
9   destruct s as (s, Hs).
10  destruct t as (t, Ht).
11  unfold bv_ult, bv_slt, bv_ashr_a, bv_eq, bv in *. cbn in *.
12  specialize (InvCond.bvashr_ult2_rtl n s t Hs Ht); intro STIC.
13  rewrite Hs, Ht in STIC. apply STIC.
14  now exists x.
15 Qed.
```

Fig. 5. A proof of one direction of the invertibility equivalence for $\gg_a$ and $<_u$ using dependent types.

The new definition uses `firstn` and `++`, over which many useful properties are already proven in the standard library. This benefits us in manual proofs, and in calls to CoqHammer, since the latter is able to use lemmas from the imported libraries to prove the goals that are given to it. Using this representation, proving Eq. (6) reduces to proving Lemmas `bv_ule_1_firstn` and `bv_ule_pre_append`, shown in Fig. 6. The proof of `bv_ule_pre_append` benefited from the property `app_comm_cons` from the standard list library of Coq, whereas `firstn_length_le` was useful in reducing the goal of `bv_ule_1_firstn` to the Coq equivalent of Eq. (3). The statements of the properties mentioned from the standard library are also shown in Fig. 6. □

Finally, we examine what was considered a challenge problem in the previous version of this work [11]. The next example details how we completed the proof.

Example 5. Consider the literal $(x \gg s) >_u t$. Its invertibility condition is $t <_u (\sim s \gg s)$. Now consider the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ t <_u (\sim s \gg s) \Rightarrow \exists x : \sigma_{[n]}.\ (x \gg s) >_u t \quad (7)$$

Figure 7 contains the theorem stating the equivalence, and some lemmas used within its proof. A crucial step in the proof of the implication is to rewrite the definition of the right shift operator `bv_shr` to its alternate definition `bv_shr_a` (see Sect. 4.2). Unfolding the alternative definition leads to a case-analysis on the following condition:

$$\mathtt{toNat}(s) < \mathtt{len}(x)$$

where `toNat` casts a bit-vector to its natural number representation, and `len` returns the length of a bit-vector as a natural number.

```
1  Lemma bv_ule_1_firstn : forall (n : nat) (x : bitvector),
2   (n < length x)%nat ->
3   bv_ule (firstn n x) firstn n (mk_list_true (length x))) = true.
4
5  Lemma bv_ule_pre_append : forall (x y z : bitvector),
6   bv_ule x y  = true -> bv_ule (z ++ x) (z ++ y) = true.
7
8  Theorem app_comm_cons : forall (x y:list A) (a:A),
9   a ::  (x ++ y) = (a :: x) ++ y
10
11 Lemma firstn_length_le: forall l:list A, forall n:nat,
12  n <= length l -> length (firstn n l) = n.
```

Fig. 6. Examples of lemmas used in proofs of invertibility equivalences.

The challenge in the proof arises in the positive case of the condition, which reduces to a proof of `first_bits_zero` (see Fig. 7). `first_bits_zero` says that given $\mathtt{toNat}(s) < \mathtt{len}(s)$, the most-significant $\mathtt{len}(s) - \mathtt{toNat}(s)$ bits of s are 0. As seen in Fig. 4, the second argument to the top-most layer of the shift (called from `bv_shl_eq`) is a bit-vector that specifies the number of times to shift the bit-vector in the first argument. This second argument is converted to a natural number by the abstract `toNat` function invoked above, the concrete definitions of which are specified in Fig. 7 as `list2nat_be_a` and `list2N`. At the same level of abstraction, we use `rev` for the list reversal function corresponding to the Coq function of the same name, and `firstn` also for its Coq namesake (`firstn` n l returns the n most significant bits of l), so that `first_bits_zero` can be specified as follows:

$$\mathtt{toNat}(s) < \mathtt{len}(s) \Rightarrow \mathtt{firstn}\ (\mathtt{len}(s) - \mathtt{toNat}(s))\ (\mathtt{rev}(s)) = 0$$

The intuition behind its validity is that if the most-significant $\mathtt{len}(s) - \mathtt{toNat}(s)$ bits were not 0 then they would contribute to the value of $\mathtt{toNat}(s)$, making it

greater than or equal to `len`(s) and thus falsifying the condition. However, it is challenging to convert this intuition into a proof using induction over lists, as explained in what follows.

To prove `first_bits_zero`, we redefined `list2N` as a tail-recursive function `list2NTR`. This step was proven to be sound by a lemma of equivalence between the two definitions (`list2N_eq`). Since `list2N` is not tail recursive, it only begins computation at the end of the input list representing a bit-vector. Such a definition further complicates the proof of `first_bits_zero` when based on the typical induction principle over the structure of the Boolean list underlying the bit-vector `s`. This is because it does not easily reduce (via ι-reduction for inductive definitions [19]), into a useful expression in the step case of the intended induction.

The advantage of tail recursion in this context is best illustrated by Fig. 8 where `x` is a Boolean variable and `xs` represents an arbitrary Boolean list. The

```
1   Theorem bvshr_ugt_ltr : forall (n : N), forall (s t : bitvector n),
2        (bv_ult t (bv_shr (bv_not s) s) = true) ->
3      (exists (x : bitvector n), bv_ugt (bv_shr x s) t = true).
4
5  Lemma first_bits_zero : forall (s : bitvector),
6   (N.to_nat (list2N s) < length s)%nat ->
7   firstn (length s - N.to_nat (list2N s)) (rev s) =
8   mk_list_false (length s -N.to_nat (list2N s)).
9
10 Lemma first_bits_zeroA : forall (s : bitvector),
11  (length s >= (list2NTR s))%nat ->
12  firstn (length s - (list2NTR s)) s =
13  mk_list_false (length s -(list2NTR s)).
14
15 Fixpoint list2N (a: list bool) :=
16  match a with
17   | [ ] => 0
18   | x ::   xs => if x then N.succ_double (list2N xs) else
19                  N.double (list2N xs)
20  end.
21
22 Definition list2nat_be_a (a: list bool) := N.to_nat (list2N a).
23
24 Fixpoint list2NR (a: list bool) (n: nat) :=
25  match a with
26   | [ ]      => n
27   | x ::  xs => if x then list2NR xs (2 * n + 1) else
28                 list2NR xs (2 * n)
29 end.
30
31 Definition list2NTR (a: list bool) := list2NR a 0.
32
33 Lemma list2N_eq: forall (s: bitvector),
34  list2NTR (rev s) = N.to_nat (list2N s).
```

Fig. 7. Invertibility equivalence for $\gg$ and $>_u$ and some lemmas used by its proof.

$$\frac{x\texttt{: bool} \quad xs\texttt{: list bool} \quad \texttt{IH: firstn } (\texttt{len}(xs) - \texttt{toNat}(xs))\ (\texttt{rev}(xs)) = 0}{\texttt{Goal: firstn } (\texttt{len}(xs) + 1 - \texttt{toNat}(x :: xs))\ (\texttt{rev}(x :: xs)) = 0} \quad (8)$$

$$\frac{x\texttt{: bool} \quad xs\texttt{: list bool} \quad \texttt{IH: firstn } (\texttt{len}(xs) - \texttt{toNatTR}(xs))\ (xs) = 0}{\texttt{Goal: firstn } (\texttt{len}(xs) + 1 - \texttt{toNatTR}(xs \texttt{ ++ } [x]))\ (xs \texttt{ ++ } [x]) = 0} \quad (9)$$

Fig. 8. Sub-goals generated in the proof of `first_bits_zero`. Note that 0 is a bit-vector constant of the appropriate length (list of `false`s).

derivation of the goal from the inductive hypothesis (`IH`) in derivation (8) from Fig. 8 is complicated in Coq because the functions `firstn` and `rev` are not well-matched with `list2N`, if not incompatible. For instance, observe that the in the inductive step (`Goal`), as the first argument to `firstn` increases, the number of bits fetched from the list increases towards the *right*. However, due to the little-endian notation of bit-vectors and the fact that the list `cons` function (`::`) can be seen as incrementing its argument list to its *left*, the `rev` function must be used to corrects the direction of increase of the second argument to `firstn`. Despite this correction, an induction over s must deal with two structurally different lists.

In contrast, the tail-recursive definition of `list2NTR` hides the `rev` function. This is illustrated in derivation (9) in Fig. 8, where `toNatTR` corresponds to `list2NTR`. Furthermore, such an induction over lists using `append` (`++`) to the right, rather than `cons` to the left is possible thanks to the *reverse induction principle*[2]. Closing such a goal allowed us to prove the `list2NTR`-variant of `first_bits_zero`, specified as `first_bits_zeroA` in Fig. 7, and the proof of equivalence between the two definitions (`list2N_eq`) allowed us to use this in closing the original goal (7). □

5.3 Results

Table 2 summarizes the results of proving invertibility equivalences for invertibility conditions in the signature Σ_0. In the table, ✓ means that the invertibility equivalence was successfully verified in Coq but not in Niemetz et al. [17], and ✓ means the opposite; ✓✓ means that the invertibility equivalence was verified using both approaches. We successfully proved all invertibility equivalences over $=$ that are expressible in Σ_0, including 4 that were not proved in [17]. For the rest of the predicates, we focused only on the 8 invertibility equivalences that were not proved in [17], and succeeded in proving all of them.

Our work thus complements [17] in verifying all invertibility conditions in Σ_0 for arbitrary bit-widths, by proving all 12 equivalences that were previously unverified, and corroborating 7 others that were verified by SMT solvers. It also complements [15], which verified all invertibility conditions in Σ_1, but only up to bit-width of 65.

[2] see `rev_ind` in https://coq.inria.fr/library/Coq.Lists.List.html.

Table 2. Proved invertibility equivalences in Σ_0 where $\bowtie$ ranges over the given predicate symbols. ✔ means that the invertibility equivalence was successfully verified in Coq but not in [17], whereas ✓ means the opposite; ✔✓ means that the invertibility equivalence was verified using both approaches.

$\ell[x]$	$=$	$\neq$	$<_u$	$>_u$	$\leq_u$	$\geq_u$
$-x \bowtie t$	✔✓	✓	✓	✓	✓	✓
$\sim x \bowtie t$	✔✓	✓	✓	✓	✓	✓
$x \,\&\, s \bowtie t$	✔	✓	✓	✓	✓	✓
$x \mid s \bowtie t$	✔	✓	✓	✓	✓	✓
$x \ll s \bowtie t$	✔	✔	✓	✔	✓	✔
$s \ll x \bowtie t$	✔✓	✓	✓	✓	✓	✓
$x \gg s \bowtie t$	✔✓	✓	✓	✔	✓	✓
$s \gg x \bowtie t$	✔✓	✓	✓	✓	✓	✓
$x \gg_a s \bowtie t$	✔	✓	✓	✓	✓	✓
$s \gg_a x \bowtie t$	✔✓	✓	✔	✔	✔	✔
$x + s \bowtie t$	✔✓	✓	✓	✓	✓	✓

6 Conclusion and Future Work

We have described our work on verifying bit-vector invertibility conditions in the Coq proof assistant, which required extending the `BVList` library in Coq. In addition to describing the library and our extensions to it, this paper presented details about the Coq proofs of the invertibility equivalences. These were done on a representative subset of the operators from the theory of bit-vectors that is well-supported by the extended library. We were able to prove in Coq all the equivalences that were left unproven in previous attempts for all bit-widths, and also to prove in Coq some equivalences that were proven automatically before, thus increasing confidence in their correctness.

The most immediate direction for future work is proving more of the invertibility equivalences supported by the bit-vector library. In addition, we plan to extend the library so that it supports the full syntax in which invertibility conditions are expressed, namely Σ_1. This will also increase the potential usage of the library for other applications. Another direction for future work is to extend the proofs for invertibility conditions where some of the bits are known. Such invertibility conditions were introduced by Niemetz and Preiner [14]. However, their formal verification for every bit-width is yet to be done.

Acknowledgements. This work was funded in part by NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF), and ISF grant number 619/21.

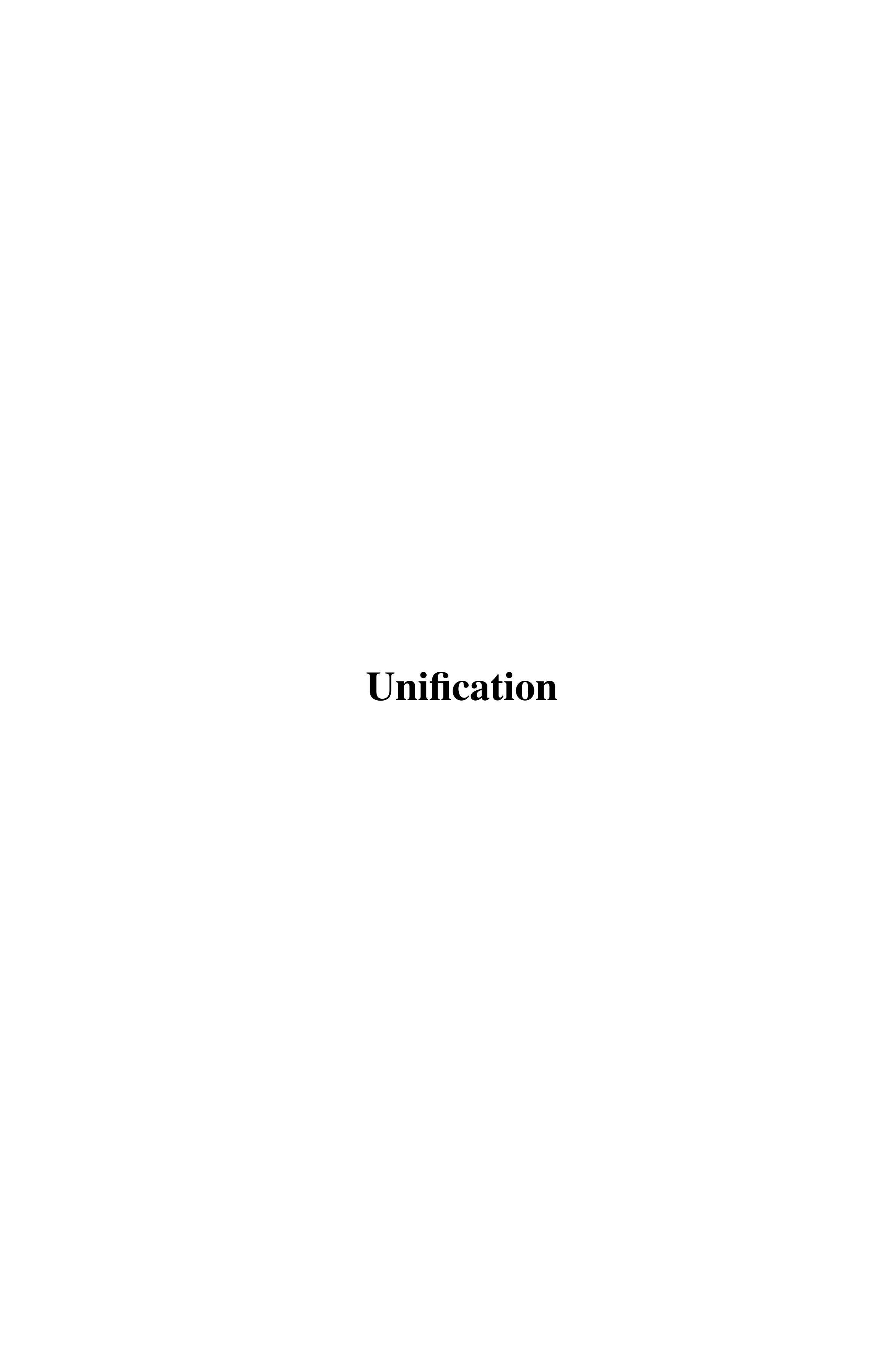
References

1. Harrison, J.: A HOL theory of euclidean space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_8
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., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: A. Gupta & D. Kroening, editors: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Beeren, J., et al.: Finite Machine Word Library. Archive of Formal Proofs. https://isa-afp.org/entries/WordLib.html Formal proof development (2016)
5. Blot, A., Dagand, P.É., Lawall, J.: Bit Sequences and Bit Sets Library. Available at https://github.com/pedagand/ssrbit
6. Chajed, T., et al.: Bedrock Bit Vectors Library. Available at https://github.com/mit-plv/bbv
7. Czajka, L., Kaliszyk, C.: Hammer for Coq: automation for dependent type theory. J. Autom. Reason. **61**(1-4), pp. 423–453 (2018). https://doi.org/10.1007/s10817-018-9458-4
8. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNAI, vol. 1955, pp. 85–95. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44404-1_7
9. Duprat, J.: Library Coq. Bool. Bvector. https://coq.inria.fr/library/Coq.Bool.Bvector.html
10. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_7
11. Ekici, B., Viswanathan, A., Zohar, Y., Barrett, C.W., Tinelli, C.: Verifying Bit-vector Invertibility Conditions in Coq (Extended Abstract). In Giselle Reis & Haniel Barbosa, editors: Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019. EPTCS **301**, pp. 18–26 (2019). https://doi.org/10.4204/EPTCS.301.4. Available at https://doi.org/10.4204/EPTCS.301.4
12. Herbert, B. Enderton (2001): Chapter TWO - First-Order Logic. In Herbert B. Enderton, editor: A Mathematical Introduction to Logic (Second Edition), second edition edition, Academic Press, Boston, pp. 67–181, https://doi.org/10.1016/B978-0-08-049646-7.50008-4
13. Gupta, A., Fisher, A.L.: Representation and symbolic manipulation of linearly inductive boolean functions. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design, ICCAD '93, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 192–199 (1993). Available at http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=259794.259827
14. Niemetz, A., Preiner, M.: Ternary Propagation-Based Local Search for more Bit-Precise Reasoning. In: FMCAD, IEEE, pp. 214–224 (2020)
15. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16

16. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. J. Autom. Reason. **65**(7), 1001–1025 (2021). https://doi.org/10.1007/s10817-021-09598-9
17. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in SMT Solvers. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 366–384. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_22
18. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): 5. the rules of the game. In: Isabelle/HOL. LNCS, vol. 2283, pp. 67–104. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9_5
19. Paulin-Mohring, C.: Inductive definitions in the system Coq rules and properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 328–345. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037116
20. Paulin-Mohring, C.: Introduction to the Calculus of Inductive Constructions. In: Bruno Woltzenlogel Paleo & David Delahaye, editors: All about Proofs, Proofs for All, Studies in Logic (Mathematical logic and foundations) 55, College Publications. https://hal.inria.fr/hal-01094195 (2015)
21. Paulsson, 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, October 9, 2011, EPiC Series in Computing 2, EasyChair, pp. 1–11, https://doi.org/10.29007/36dt. Available at https://doi.org/10.29007/36dt
22. Shi, X., Fu, Y.-F., Liu, J., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: CoqQFBV: a scalable certified SMT quantifier-free bit-vector solver. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 149–171. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_7
23. Sozeau, M.: Equations: A dependent pattern-matching compiler. In: Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP 2010), pp. 419–434 (2010). https://doi.org/10.1007/978-3-642-14052-5_29
24. Spies, S., Forster, Y.: Undecidability of higher-order unification formalised in Coq. In: Blanchette, J., Hritcu, C., eds.: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020, ACM, pp. 143–157, https://doi.org/10.1145/3372885.3373832. Available at https://doi.org/10.1145/3372885.3373832
25. The Coq development team (2019): The Coq Proof Assistant Reference Manual Version 8.9. Available at https://coq.inria.fr/distrib/current/refman/

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.



Unification

Weighted Path Orders Are Semantic Path Orders

Teppei Saito(✉) and Nao Hirokawa

JAIST, Nomi, Japan
{saito,hirokawa}@jaist.ac.jp

Abstract. We explore the relationship between weighted path orders and (monotonic) semantic path orders. Our findings reveal that weighted path orders can be considered instances of a variant of semantic path orders that comprise order pairs. This observation leads to a generalization of weighted path orders that does not impose simplicity on their underlying algebras. As a result, the generalized version is capable of proving termination of term rewrite systems beyond the realm of simple termination. In order to assess practicality we provide experimental data comparing generalized weighted path orders with the original ones as well as other well-known classes of reduction orders.

Keywords: Term Rewriting · Termination · Weighted Path Order · Semantic Path Order

1 Introduction

Reduction orders are a fundamental tool in termination analysis of term rewrite systems, and they also underlie completion-based automated theorem proving. *Weighted path orders* (WPOs) [27] are known as a versatile class of reduction orders; WPOs can simulate (generalized) Knuth–Bendix orders [7,13,16] and lexicographic path orders [12], depending on the choice of parameters, namely *simple monotone algebras* and precedences. In fact, weighted path orders are so powerful that they characterize simple termination of term rewrite systems [20, Definition 6.3.7], that is, a term rewrite system is simply terminating if and only if it admits a compatible WPO. Besides automated termination analysis [14,26], WPOs are used in reachability analysis [25], and automated theorem proving [11,18].

Another well-known class of reduction orders is the class of *monotonic semantic path orders* (MSPOs) [4,5], which are a monotonic version of semantic path orders (SPOs) [12]. MSPOs take triples of orders (called reduction triples) as parameters, and provide a complete characterization of terminating term rewrite

T. Saito—supported by JST SPRING, Grant Number JPMJSP2102. N. Hirokawa—supported by JSPS KAKENHI Grant Number JP22K11900.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 63–80, 2023.
https://doi.org/10.1007/978-3-031-43369-6_4

systems: A term rewrite system is terminating if and only if it admits a compatible MSPO. However, the relationship between WPOs and MSPOs has not been known [24].

In this paper, we give a solution to the open problem, demonstrating an effective construction of an MSPO from the algebra and the precedence of a given WPO. The key of the proof lies in finding a suitable new *variant* of MSPOs, which is described as follows: First, the variant uses lexicographic comparison [4, Definition 4.5.1], as the original WPOs [27, Definition 5] are based on this comparison strategy. Second, the variant employs reduction triples [4, Definition 4.1.19] because an example shows that a variant based on (quasi-)reduction pairs [5, Definition 4] leads to an invalid construction.

The obtained simulation result leads to a generalization of WPOs that does not impose simplicity on their underlying algebras. The generalization can show termination of term rewrite systems that are not simply terminating. This is a sharp contrast to the termination proving power of WPOs. In addition, upgrading WPOs to GWPOs can be done with little implementation effort, so we anticipate that tools which employ WPOs as reduction orders (e.g. [11,14,18,23,25,26]) may benefit from power of GWPOs.

The remaining part of the paper is organized as follows. After recalling notions and notations for term rewriting and WPOs in Sect. 2, we introduce a slightly modified version of semantic path orders that employs order pairs in Sect. 3. In Sect. 4 we show that weighted path orders are instances of semantic path orders. Using this fact, we introduce a generalization of WPOs in Sect. 5. In Sect. 6 experimental data for (generalized) weighted path orders are reported. As in the case of MSPOs [5, Section 5.2], GWPOs are capable of simulating a basic version of the dependency pair method [1]. This is discussed in Sect. 7. The paper is concluded by stating related work in Sect. 8.

2 Preliminaries

Throughout the paper, we assume familiarity with term rewriting [3,20]. First we briefly recall basic notions for term rewriting and reduction orders, and then introduce weighted path orders.

2.1 Term Rewriting

Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a countable set of variables with $\mathcal{F} \cap \mathcal{V} = \varnothing$. The set of all *terms* built from $\mathcal{F}$ and $\mathcal{V}$ is referred to as $\mathcal{T}(\mathcal{F}, \mathcal{V})$, or just as $\mathcal{T}$ when $\mathcal{F}$ and $\mathcal{V}$ are clear from the context. When we need to indicate the arity n of a function symbol f, we write $f^{(n)}$ for f. Quasi-orders on the signature are called *(quasi-)precedences*. A quasi-precedence $\succsim$ is called *well-founded* if its strict part $\succ$ is well-founded. The *size* $|t|$ of a term t is the number of function symbols and variables occurring in t. Let $\square$ be a constant with $\square \notin \mathcal{F}$. *Contexts* are terms over $\mathcal{F} \cup \{\square\}$ that contain exactly one $\square$. The term resulting from replacing $\square$ in a context C by a term t is denoted by $C[t]$. We write $s \trianglerighteq t$ if there is a context C

with $s = C[t]$. The strict part of $\trianglerighteq$ is denoted by $\triangleright$. A *substitution* is a mapping σ from variables to terms such that $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. The application $t\sigma$ of a substitution σ to a term t is inductively defined as follows: $t\sigma = \sigma(t)$ if t is a variable, and $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ if $t = f(t_1, \ldots, t_n)$.

A pair (ℓ, r) of terms is said to be a *rewrite rule* if ℓ is not a variable and every variable in r occurs in ℓ. Rewrite rules (ℓ, r) are written by $\ell \to r$. A set of rewrite rules is called a *term rewrite system* (TRS). Let $\mathcal{R}$ be a TRS. We write $\mathcal{D}_\mathcal{R}$ for the set of *defined symbols* $\{f \mid f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}\}$. The relation $\to_\mathcal{R}$ is defined on terms as follows: $s \to_\mathcal{R} t$ if there exist a rewrite rule $\ell \to r \in \mathcal{R}$, a context C, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$ hold. The TRS $\mathcal{R}$ is said to be *terminating* if there is no infinite sequence $t_1 \to_\mathcal{R} t_2 \to_\mathcal{R} \cdots$. A relation $\rightsquigarrow$ on terms is *closed under contexts* if $C[s] \rightsquigarrow C[t]$ holds whenever $s \rightsquigarrow t$ and C is a context, and it is called *closed under substitutions* or just *stable* if $s\sigma \rightsquigarrow t\sigma$ holds whenever $s \rightsquigarrow t$ and σ is a substitution. We say $\rightsquigarrow$ has the *subterm property* if $s \rightsquigarrow t$ for all terms s, t satisfying $s \triangleright t$. Relations closed under contexts and substitutions are called *rewrite relations*.

Termination is often shown by using orders. We say that a rewrite relation is a *rewrite preorder* or *reduction order* if it is a preorder or a well-founded order, respectively. A TRS $\mathcal{R}$ is *compatible* with a strict order $>$ if $\mathcal{R} \subseteq >$.

Proposition 1. *A TRS $\mathcal{R}$ is terminating if $\mathcal{R}$ is compatible with some reduction order $>$.*

An *ordered $\mathcal{F}$-algebra* is a triple $(A, \{f_\mathcal{A}\}_{f \in \mathcal{F}}, >)$, where A is a set called a *carrier*, $f_\mathcal{A}$ is an n-ary function on A (called an *interpretation function*) associated with each $f^{(n)} \in \mathcal{F}$, and $>$ is a strict order on A. Let $\mathcal{A} = (A, \{f_\mathcal{A}\}_{f \in \mathcal{F}}, >)$ be an ordered algebra. A mapping from $\mathcal{V}$ to A is called an *assignment* for $\mathcal{A}$. The interpretation $[\alpha]_\mathcal{A}(t)$ of a term t under an assignment α is inductively defined as follows: $[\alpha]_\mathcal{A}(t) = \alpha(t)$ if t is a variable, and $[\alpha]_\mathcal{A}(t) = f_\mathcal{A}([\alpha]_\mathcal{A}(t_1), \ldots, [\alpha]_\mathcal{A}(t_n))$ if $t = f(t_1, \ldots, t_n)$. We write $s >_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) > [\alpha]_\mathcal{A}(t)$ for all assignments α. The relation $>_\mathcal{A}$ is a strict order. Similarly we write $s \geqslant_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) \geqslant [\alpha]_\mathcal{A}(t)$ holds for all assignments α, where $\geqslant$ stands for the reflexive closure of $>$. The relation $\geqslant_\mathcal{A}$ is a quasi-order, and satisfies $\geqslant_\mathcal{A} \cdot >_\mathcal{A} \cdot \geqslant_\mathcal{A} \subseteq >_\mathcal{A}$. We say that the ordered algebra $\mathcal{A}$ is

- *simple* if $f_\mathcal{A}(a_1, \ldots, a_i, \ldots, a_n) \geqslant a_i$ for all $f^{(n)} \in \mathcal{F}$, $1 \leqslant i \leqslant n$, and $a_1, \ldots, a_n \in A$;
- *weakly monotone* if $f_\mathcal{A}(a_1, \ldots, a_i, \ldots, a_n) \geqslant f_\mathcal{A}(a_1, \ldots, b, \ldots, a_n)$ for all $f^{(n)} \in \mathcal{F}$, argument positions $1 \leqslant i \leqslant n$, and $a_1, \ldots, a_n, b \in A$ with $a_i > b$;
- *simple monotone* if it is simple and weakly monotone; and
- *well-founded* if $>$ is well-founded.

If $>$ is well-founded, so is $>_\mathcal{A}$. If $\mathcal{A}$ is a weakly monotone algebra, $\geqslant_\mathcal{A}$ is a rewrite preorder. If in addition $\mathcal{A}$ is simple, $\geqslant_\mathcal{A}$ has the subterm property $\triangleright \subseteq \geqslant_\mathcal{A}$.

2.2 Weighted Path Orders

Weighted path orders (WPOs) are reduction orders introduced by Yamada et al. [27]. The definition of WPOs is based on the pair of an ordered algebra $\mathcal{A}$

and a precedences $\succsim$. A WPO compares terms s, t as a generalized KBO does: First the terms are compared by $s >_{\mathcal{A}} t$. If only weak inequality $s \geqslant_{\mathcal{A}} t$ holds then their root symbols, say f and g, are compared by the precedence $\succsim$. If again only weak inequality $f \succsim g$ holds, arguments are compared *lexicographically*.

Lexicographic comparison is formalized as follows. Let $>$ be a strict order on a set A and let A^* denote the set of all strings (tuples) over A. The *lexicographic extension* $>^{\mathsf{lex}}$ of $>$ is defined on A^* as follows: $(a_1, \ldots, a_n) >^{\mathsf{lex}} (b_1, \ldots, b_m)$ if there is a natural number $k < n$ such that

– $a_j = b_j$ for all $1 \leqslant j \leqslant k$, and
– either $k = m$, or $k < m$ and $a_{k+1} > b_{k+1}$.

It is known that $>^{\mathsf{lex}}$ is a strict order on A^*.

Definition 1 ([27]). *Let $\mathcal{A}$ be an ordered $\mathcal{F}$-algebra and $\succsim$ a precedence. The* weighted path order $>_{\mathsf{wpo}}$ *is defined on terms over $\mathcal{F}$ as follows: $s >_{\mathsf{wpo}} t$ if*

1. $s >_{\mathcal{A}} t$, or
2. $s \geqslant_{\mathcal{A}} t$, $s = f(s_1, \ldots, s_m)$, and one of the following conditions holds.
   a. $s_i \geqslant_{\mathsf{wpo}} t$ for some $1 \leqslant i \leqslant m$.
   b. $t = g(t_1, \ldots, t_n)$ and $s >_{\mathsf{wpo}} t_j$ for all $1 \leqslant j \leqslant n$, and moreover
      (i) $f \succ g$, or
      (ii) $f \succsim g$ and $(s_1, \ldots, s_m) >^{\mathsf{lex}}_{\mathsf{wpo}} (t_1, \ldots, t_n)$.

Here $\geqslant_{\mathsf{wpo}}$ denotes the reflexive closure of $>_{\mathsf{wpo}}$.

Theorem 1 ([27]). *Suppose that the signature is finite. For every simple monotone well-founded algebra and well-founded precedence the induced relation $>_{\mathsf{wpo}}$ is a reduction order with the subterm property.*

Example 1. Consider the following TRS $\mathcal{R}$ taken from [27, Example 9]:

$$\mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{f}(\mathsf{f}(x))) \qquad \mathsf{f}(\mathsf{h}(x)) \to \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))$$

Let $\mathcal{A}$ be the simple monotone algebra on $\mathbb{N}$ with $\mathsf{f}_{\mathcal{A}}(x) = \mathsf{h}_{\mathcal{A}}(x) = x$ and $\mathsf{g}_{\mathcal{A}}(x) = x + 1$. Take a precedence $\succsim$ with $\mathsf{f} \succ \mathsf{g} \succ \mathsf{h}$. The relation $\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is verified by the following derivation:

$$\dfrac{\mathsf{f}(\mathsf{g}(x)) \geqslant_{\mathcal{A}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x))) \qquad \mathsf{f} \succ \mathsf{g} \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) >_{\mathcal{A}} \mathsf{f}(\mathsf{f}(x))}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{f}(\mathsf{f}(x))}\ \text{WPO 1}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{wpo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))}\ \text{WPO 2b(i)}$$

Here WPO 1 and WPO 2b(i) indicate the corresponding conditions in Definition 1. Similarly, one can verify $\mathsf{f}(\mathsf{h}(x)) >_{\mathsf{wpo}} \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))$. Therefore, $\mathcal{R} \subseteq >_{\mathsf{wpo}}$ follows. Hence, we conclude that $\mathcal{R}$ is terminating.

The following example shows that the simplicity condition cannot be dropped from Theorem 1.

Example 2. Any WPO $>_{\mathsf{wpo}}$ induced by the weakly monotone but non-simple algebra $\mathcal{A}$ on $\mathbb{N}$ with $\mathsf{a}_{\mathcal{A}} = 1$ and $\mathsf{f}_{\mathcal{A}}(x) = 0$ lacks well-foundedness as it admits the cyclic sequence $\mathsf{f}(\mathsf{a}) >_{\mathsf{wpo}} \mathsf{f}(\mathsf{f}(\mathsf{a})) >_{\mathsf{wpo}} \mathsf{f}(\mathsf{a})$.

3 Semantic Path Orders Based on Order Pairs

Borralleras [4, Definition 4.1.19] introduced a variant of SPO that employs a pair of a quasi-order and a strict order. This variant compares arguments of terms by a multiset order. In order to simulate WPOs which compare arguments in a lexicographic manner, we introduce another variant of SPO.

We say that the pair $(\gtrsim, >)$ of a quasi-order $\gtrsim$ and a strict order $>$ is an *order pair* if $\gtrsim \cdot > \cdot \gtrsim \subseteq >$. The inclusion is referred to as *compatibility*. We say that an order pair $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ on terms is *stable* if both $\mathrel{\underset{\sim}{\sqsupset}}$ and $\sqsupset$ are stable.

Definition 2. *Let* $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ *be a stable order pair on* $\mathcal{T} \setminus \mathcal{V}$.[1] *The* semantic path order $>_{\mathsf{spo}}$ (SPO) *is defined on terms as follows:* $s >_{\mathsf{spo}} t$ *if* $s = f(s_1, \ldots, s_m)$ *and one of the following conditions hold:*

1. $s_i \geqslant_{\mathsf{spo}} t$ *for some* $1 \leqslant i \leqslant m$.
2. $t = g(t_1, \ldots, t_n)$ *and* $s >_{\mathsf{spo}} t_j$ *for all* $1 \leqslant j \leqslant n$, *and moreover*
   a. $s \sqsupset t$, *or*
   b. $s \mathrel{\underset{\sim}{\sqsupset}} t$ *and* $(s_1, \ldots, s_m) >^{\mathsf{lex}}_{\mathsf{spo}} (t_1, \ldots, t_n)$.

Here $\geqslant_{\mathsf{spo}}$ *denotes the reflexive closure of* $>_{\mathsf{spo}}$.

Remark 1. The standard definitions of SPOs ([12] and [4, Definition 4.1.19]) use the multiset extension of $>_{\mathsf{spo}}$ in SPO 2b instead of the lexicographic extension. The lexicographic version of SPOs, introduced by Borralleras [4, Definition 4.5.1], can be obtained by setting $\sqsupset$ to the strict part of $\mathrel{\underset{\sim}{\sqsupset}}$ in Definition 2.

Example 3. Lexicographic path orders (LPOs) are special instances of SPOs. Let $\succsim$ be a precedence. Define $f(s_1, \ldots, s_m) \mathrel{\underset{\sim}{\sqsupset}} g(t_1, \ldots, t_n)$ by $f \succsim g$, and let $\sqsupset$ be the strict part of $\mathrel{\underset{\sim}{\sqsupset}}$. The semantic path order induced by $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ is the lexicographic path order induced by $\succsim$.

Let $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ be a stable order pair on $\mathcal{T} \setminus \mathcal{V}$ and let $>_{\mathsf{spo}}$ be the semantic path order induced by $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$. The transitivity, reflexivity, and stability of $>_{\mathsf{spo}}$ are straightforward. A small remark is that the compatibility $\mathrel{\underset{\sim}{\sqsupset}} \cdot \sqsupset \cdot \mathrel{\underset{\sim}{\sqsupset}} \subseteq \sqsupset$ is used in the proof of the transitivity.

Lemma 1. *The SPO* $>_{\mathsf{spo}}$ *is a stable strict order.* □

When the signature is infinite, the lexicographic version of SPOs is not well-founded in general even if $\sqsupset$ is well-founded. This forms a contrast to the multiset versions of SPOs mentioned in Remark 1.

Example 4. Consider the signature consisting of $\mathsf{a}^{(0)}$, $\mathsf{b}^{(0)}$, and $\mathsf{f}_i^{(i)}$ for all numbers $i \in \mathbb{N}$. Let $\succsim$ be a well-founded precedence satisfying $\mathsf{a} \succ \mathsf{b}$ and $\mathsf{f}_i \succsim \mathsf{f}_j$ for all $i, j \in \mathbb{N}$. The pair $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ defined as in Example 3 is an order pair with $\sqsupset$ well-founded, but the SPO $>_{\mathsf{spo}}$ induced from $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$ admits the infinite chain:

$$\mathsf{f}_1(\mathsf{a}) >_{\mathsf{spo}} \mathsf{f}_2(\mathsf{b}, \mathsf{a}) >_{\mathsf{spo}} \mathsf{f}_3(\mathsf{b}, \mathsf{b}, \mathsf{a}) >_{\mathsf{spo}} \cdots$$

See [22, Section 3] and [19, Section 3] for related discussions.

---

[1] The restriction to $\mathcal{T} \setminus \mathcal{V}$ is not essential but meant to be a minimum requirement. Observe that Definition 2 uses the order pair only when s and t are not variables.

Well-foundedness of $>_{\mathsf{spo}}$ is restored by assuming existence of an upper bound of arities. We refer to this property as *boundedness* of the signature. Needless to say, a signature is bounded whenever it is finite.

Hereafter we assume that $\sqsupset$ is well-founded and $\mathcal{F}$ is bounded. For showing that $>_{\mathsf{spo}}$ is well-founded, we adopt Buchholz's method [6]. One can find a similar proof in [27, Lemma 8]. We write $\mathsf{SN}(>_{\mathsf{spo}})$ for the set of all terms t such that there is no infinite descending sequence $t >_{\mathsf{spo}} t_1 >_{\mathsf{spo}} t_2 >_{\mathsf{spo}} \cdots$ starting from t.[2] The following properties are immediate:

- The restriction of $>_{\mathsf{spo}}$ to $\mathsf{SN}(>_{\mathsf{spo}})$ is a well-founded order on $\mathsf{SN}(>_{\mathsf{spo}})$.
- $t \in \mathsf{SN}(>_{\mathsf{spo}})$ if $u \in \mathsf{SN}(>_{\mathsf{spo}})$ for all terms u with $t >_{\mathsf{spo}} u$.

Buchholz's method proves well-foundedness by well-founded induction. To express our well-founded order for induction, we recall the notion of the lexicographic product of order pairs. Let $(\gtrsim_1, >_1), \ldots, (\gtrsim_n, >_n)$ be n order pairs on sets $A_1, \ldots, A_n$, respectively. The *lexicographic product* $(\gtrsim_1, >_1) \otimes \cdots \otimes (\gtrsim_n, >_n)$ is the strict order $>$ defined on $A_1 \times \cdots \times A_n$ as follows: $(a_1, \ldots, a_n) > (b_1, \ldots, b_n)$ if there exists an index $k \in \{1, \ldots, n\}$ such that $a_k >_k b_k$ and $a_j \gtrsim_j b_j$ for all $1 \leqslant j < k$. Note that the lexicographic product $>$ is well-founded if every $>_i$ is well-founded.

Given a set A, we write $A^{\leqslant k}$ for the union of A^i for all $i \leqslant k$. If a strict order $>$ on A is well-founded, then the restriction of $>^{\mathsf{lex}}$ to $A^{\leqslant k}$ is also well-founded, see [19, Section 3]. Thus, the lexicographic product $\gg$ given by

$$(\sqsupseteq, \sqsupset) \otimes (\geqslant^{\mathsf{lex}}_{\mathsf{spo}}, >^{\mathsf{lex}}_{\mathsf{spo}}) \otimes (\trianglerighteq, \triangleright)$$

is a well-founded order on $(\mathcal{T} \setminus \mathcal{V}) \times \mathcal{T}^{\leqslant M} \times \mathcal{T}$. Here M stands for the maximum arity in the signature $\mathcal{F}$, and $\geqslant^{\mathsf{lex}}_{\mathsf{spo}}$ for the reflexive closure of $>^{\mathsf{lex}}_{\mathsf{spo}}$.

Lemma 2. *The term u belongs to $\mathsf{SN}(>_{\mathsf{spo}})$ whenever $t = f(t_1, \ldots, t_n) >_{\mathsf{spo}} u$ and $t_1, \ldots, t_n \in \mathsf{SN}(>_{\mathsf{spo}})$.*

Proof. We show the claim by well-founded induction on $(t, (t_1, \ldots, t_n), u)$ with respect to $\gg$. Here we proceed by analyzing the derivation of $t >_{\mathsf{spo}} u$. If $t >_{\mathsf{spo}} u$ is derived from SPO 1 then $t_i \geqslant_{\mathsf{spo}} u$ for some $i \in \{1, \ldots, n\}$. In this case $u \in \mathsf{SN}(>_{\mathsf{spo}})$ trivially follows from $t_i \in \mathsf{SN}(>_{\mathsf{spo}})$. If $t >_{\mathsf{spo}} u$ is derived from SPO 2a or SPO 2b, then u is of the form $g(u_1, \ldots, u_m)$ and $t >_{\mathsf{spo}} u_j$ for all $j \in \{1, \ldots, m\}$. From $u \triangleright u_j$ we have $(t, (t_1, \ldots, t_n), u) \gg (t, (t_1, \ldots, t_n), u_j)$. So from the induction hypothesis $u_i \in \mathsf{SN}(>_{\mathsf{spo}})$ for each j. For showing our goal $u \in \mathsf{SN}(>_{\mathsf{spo}})$ fix an arbitrary term v with $u >_{\mathsf{spo}} v$. We further distinguish the case of SPO 2a and that of SPO 2b.

a. If $t >_{\mathsf{spo}} u$ is derived from SPO 2a then $t \sqsupset u$. Thus, $(t, (t_1, \ldots, t_n), u) \gg (u, (u_1, \ldots, u_m), v)$, and the induction hypothesis yields $v \in \mathsf{SN}(>_{\mathsf{spo}})$.

[2] SN stands for strong normalization, which is another name of termination.

b. If $t >_{\mathsf{spo}} u$ is derived from SPO 2b then we additionally have $t \sqsupseteq u$ and $(t_1, \ldots, t_n) >^{\mathsf{lex}}_{\mathsf{spo}} (u_1, \ldots, u_m)$. Thus, $(t, (t_1, \ldots, t_n), u) \gg (u, (u_1, \ldots, u_m), v)$ holds. So from the induction hypothesis we obtain $v \in \mathsf{SN}(>_{\mathsf{spo}})$.

In either case $v \in \mathsf{SN}(>_{\mathsf{spo}})$. So we conclude $u \in \mathsf{SN}(>_{\mathsf{spo}})$. □

Lemma 3. *The relation $>_{\mathsf{spo}}$ is well-founded.*

Proof. We show that $t \in \mathsf{SN}(>_{\mathsf{spo}})$ by induction on $|t|$. If t is a variable trivially $t \in \mathsf{SN}(>_{\mathsf{spo}})$. Otherwise, Lemma 2 applies. □

Theorem 2. *Every semantic path order is a stable well-founded order, provided that the signature is bounded.* □

In general, semantic path orders are not closed under contexts. For a remedy, Borralleras et al. [5] propose the use of another preorder with the *harmony* property. This results in monotonic semantic path orders.

Definition 3 ([4, Definition 4.1.20]). *A triple $(\gtrsim, \sqsupseteq, \sqsupset)$ is a* reduction triple *if $\gtrsim$ is a rewrite preorder on terms, $(\sqsupseteq, \sqsupset)$ is a stable order pair on $\mathcal{T} \setminus \mathcal{V}$ with $\sqsupset$ well-founded, and $\gtrsim$ and $\sqsupseteq$ have the* harmony *property, meaning that for every $f^{(n)} \in \mathcal{F}$ the implication*

$$s_i \gtrsim t \implies f(s_1, \ldots, s_i, \ldots, s_n) \sqsupseteq f(s_1, \ldots, t, \ldots, s_n)$$

holds for all terms $s_1, \ldots, s_n, t$ and argument positions $1 \leqslant i \leqslant n$.

Definition 4. *Let $(\gtrsim, \sqsupseteq, \sqsupset)$ be a reduction triple, and let $>_{\mathsf{spo}}$ be the semantic path order induced from $(\sqsupseteq, \sqsupset)$. The* monotonic semantic path order $s >_{\mathsf{mspo}} t$ (MSPO) *is defined as $s \gtrsim t$ and $s >_{\mathsf{spo}} t$.*

Theorem 3. *Every monotonic semantic path order is a reduction order, provided that the signature is bounded.*

Proof. The proof due to Borralleras et al. [5, Theorem 2] goes through. □

4 Simulating WPOs by SPOs

We show that WPOs are instances of SPOs by constructing a suitable order pair $(\sqsupseteq, \sqsupset)$ from a weakly monotone well-founded algebra $\mathcal{A}$ and a well-founded precedence $\succsim$. For terms $s = f(s_1, \ldots, s_m), t = g(t_1, \ldots, t_n)$ we write $s \sqsupseteq t$ if $s >_{\mathcal{A}} t$, or both $s \geqslant_{\mathcal{A}} t$ and $f \succsim g$. Similarly, we define $s \sqsupset t$ if $s >_{\mathcal{A}} t$, or both $s \geqslant_{\mathcal{A}} t$ and $f \succ g$. It is worth noting that the proof of [27, Lemma 8] also combines the interpretation order and precedence in a lexicographic manner.

Lemma 4. *The pair $(\sqsupseteq, \sqsupset)$ is a stable order pair with $\sqsupset$ well-founded.* □

In the remaining part of the section we consider the WPO $>_{\mathsf{wpo}}$ induced by $\mathcal{A}$ and $\succsim$, and the SPO $>_{\mathsf{spo}}$ induced by the corresponding order pair $(\mathrel{\underset{\sim}{\sqsupset}}, \sqsupset)$. Note that $\sqsupset$ is not a strict part of $\mathrel{\underset{\sim}{\sqsupset}}$ in general, as $>_{\mathcal{A}}$ is not necessarily the strict part of $\geqslant_{\mathcal{A}}$. This is why we decoupled $\sqsupset$ from $\mathrel{\underset{\sim}{\sqsupset}}$ in Definition 2; see also Remark 1.

Example 5. Let the signature $\mathcal{F} = \{\mathsf{f}^{(1)}\}$. Consider the trivial precedence $\mathsf{f} \succsim \mathsf{f}$ and the algebra $\mathcal{A}$ over the carrier $\mathbb{N}$ with the interpretation $\mathsf{f}_{\mathcal{A}}(x) = 2x$. On the one hand we have $\mathsf{f}(\mathsf{f}(x)) \mathrel{\underset{\sim}{\sqsupset}} \mathsf{f}(x)$ from $\mathsf{f}(\mathsf{f}(x)) \geqslant_{\mathcal{A}} \mathsf{f}(x)$ but not $\mathsf{f}(x) \mathrel{\underset{\sim}{\sqsupset}} \mathsf{f}(\mathsf{f}(x))$ as $\mathsf{f}(x) \ngeqslant_{\mathcal{A}} \mathsf{f}(\mathsf{f}(x))$. On the other hand $\mathsf{f}(\mathsf{f}(x)) \sqsupset \mathsf{f}(x)$ does not hold.

We illustrate how the derivation of $>_{\mathsf{wpo}}$ in Example 1 is simulated by the semantic path order.

Example 6 (continued from Example 1). From $\mathsf{f}(\mathsf{g}(x)) \geqslant_{\mathcal{A}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ and $\mathsf{f} \succ \mathsf{g}$ the inequality $\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is obtained. Moreover, we have $\mathsf{f}(\mathsf{g}(x)) >_{\mathcal{A}} \mathsf{f}(\mathsf{f}(x))$. Since $\geqslant_{\mathcal{A}}$ has the subterm property, the subterm $\mathsf{f}(x)$ of $\mathsf{f}(\mathsf{f}(x))$ also satisfies $\mathsf{f}(\mathsf{g}(x)) >_{\mathcal{A}} \mathsf{f}(x)$. Thus we obtain $\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(\mathsf{f}(x)), \mathsf{f}(x)$. Therefore, $\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))$ is verified as follows:

$$\dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{g}(\mathsf{f}(\mathsf{f}(x))) \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(\mathsf{f}(x)) \qquad \dfrac{\mathsf{f}(\mathsf{g}(x)) \sqsupset \mathsf{f}(x) \qquad \dfrac{\dfrac{x \geqslant_{\mathsf{spo}} x}{\mathsf{g}(x) \geqslant_{\mathsf{spo}} x}\;\text{SPO 1}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} x}\;\text{SPO 1}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{f}(x)}\;\text{SPO 2a}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{f}(\mathsf{f}(x))}\;\text{SPO 2a}}{\mathsf{f}(\mathsf{g}(x)) >_{\mathsf{spo}} \mathsf{g}(\mathsf{f}(\mathsf{f}(x)))}\;\text{SPO 2a}$$

Similarly, $\mathsf{f}(\mathsf{h}(x)) >_{\mathsf{spo}} \mathsf{h}(\mathsf{h}(\mathsf{f}(x)))$ can be verified. Hence, the inclusion $\mathcal{R} \subseteq >_{\mathsf{spo}}$ holds. Observe that the use of WPO 1 in Example 1 is replaced by successive application of SPO 1 and SPO 2a.

As shown in the example, the subterm property of $\geqslant_{\mathcal{A}}$ is a key for filling in the gap between $>_{\mathsf{spo}}$ and $>_{\mathsf{wpo}}$.

Lemma 5. *Suppose that $\mathcal{A}$ is simple. If $s >_{\mathsf{wpo}} t$ then $s >_{\mathsf{spo}} t$.*

Proof. We prove the claim by induction on $|s| + |t|$. Let $s = f(s_1, \ldots, s_m) >_{\mathsf{wpo}} t$. Depending on the derivation of $s >_{\mathsf{wpo}} t$, we distinguish five cases.

- Suppose that t is a variable and $s >_{\mathsf{wpo}} t$ is derived from $s >_{\mathcal{A}} t$. One can verify that t occurs in s. Because s is not a variable, $s \rhd t$ follows. By the subterm property of $>_{\mathsf{spo}}$ we obtain $s >_{\mathsf{spo}} t$.
- Suppose that $t = g(t_1, \ldots, t_n)$ and $s >_{\mathsf{wpo}} t$ is derived from $s >_{\mathcal{A}} t$. From $s >_{\mathcal{A}} t$ we obtain $s \sqsupset t$. Since $\mathcal{A}$ is simple, for every $1 \leqslant j \leqslant n$ we have $s >_{\mathcal{A}} t \geqslant_{\mathcal{A}} t_j$, which leads to $s >_{\mathcal{A}} t_j$. Hence, $s >_{\mathsf{spo}} t$ is derived as follows:

$$\dfrac{s \sqsupset t \qquad \dfrac{\dfrac{\forall j.\;\; s >_{\mathcal{A}} t_j}{\forall j.\;\; s >_{\mathsf{wpo}} t_j}\;\text{WPO 1}}{\forall j.\;\; s >_{\mathsf{spo}} t_j}\;\text{I.H.}}{s >_{\mathsf{spo}} g(t_1, \ldots, t_n) = t}\;\text{SPO 2a}$$

– Suppose that $s >_{\mathsf{wpo}} t$ is derived as follows:

$$\frac{s \geqslant_{\mathcal{A}} t \qquad s_i \geqslant_{\mathsf{wpo}} t}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} t}\ \text{WPO 2a}$$

By the induction hypothesis we have $s_i \geqslant_{\mathsf{spo}} t$ for some i, and thus $s >_{\mathsf{spo}} t$.

– Suppose that $s >_{\mathsf{wpo}} t$ is derived as follows:

$$\frac{s \geqslant_{\mathcal{A}} t \qquad f \succ g \qquad \forall j.\ \ s >_{\mathsf{wpo}} t_j}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} g(t_1, \ldots, t_m) = t}\ \text{WPO 2b(i)}$$

From $s \geqslant_{\mathcal{A}} t$ and $f \succ g$ we obtain $s \sqsupset t$. Thus, we have:

$$\frac{s \sqsupset t \qquad \dfrac{\forall j.\ \ s >_{\mathsf{wpo}} t_j}{\forall j.\ \ s >_{\mathsf{spo}} t_j}\ \text{I.H.}}{s = f(s_1, \ldots, s_n) >_{\mathsf{spo}} g(t_1, \ldots, t_m) = t}\ \text{SPO 2a}$$

– Suppose that $s >_{\mathsf{wpo}} t$ is derived as follows:

$$\frac{s \geqslant_{\mathcal{A}} t \quad f \succsim g \quad \forall j.\ \ s >_{\mathsf{wpo}} t_j \quad (s_1, \ldots, s_n) >^{\mathsf{lex}}_{\mathsf{wpo}} (t_1, \ldots, t_m)}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} g(t_1, \ldots, t_m) = t}\ \text{WPO 2b(ii)}$$

From $s \geqslant_{\mathcal{A}} t$ we obtain $s \underset{\sim}{\sqsupset} t$. Thus, we have:

$$\frac{s \underset{\sim}{\sqsupset} t \qquad \dfrac{\forall j.\ \ s >_{\mathsf{wpo}} t_j}{\forall j.\ \ s >_{\mathsf{spo}} t_j}\ \text{I.H.} \qquad \dfrac{(s_1, \ldots, s_n) >^{\mathsf{lex}}_{\mathsf{wpo}} (t_1, \ldots, t_m)}{(s_1, \ldots, s_n) >^{\mathsf{lex}}_{\mathsf{spo}} (t_1, \ldots, t_m)}\ \text{I.H.}}{s = f(s_1, \ldots, s_n) >_{\mathsf{spo}} g(t_1, \ldots, t_m) = t}\ \text{SPO 2b}$$

In any case we have $s >_{\mathsf{spo}} t$. ☐

Next we prove the converse direction of Lemma 5. The next lemma is a basic property of WPOs.

Lemma 6. *If $s >_{\mathsf{wpo}} t$ then $s \geqslant_{\mathcal{A}} t$.* ☐

Lemma 7. *Suppose that $\mathcal{A}$ is simple. If $s >_{\mathsf{spo}} t$ then $s >_{\mathsf{wpo}} t$.*

Proof. We prove the claim by induction on $|s| + |t|$. We distinguish three cases, depending on the derivation of $s >_{\mathsf{spo}} t$.

– Suppose that $s >_{\mathsf{spo}} t$ is derived as follows:

$$\frac{s_i \geqslant_{\mathsf{spo}} t}{s = f(s_1, \ldots, s_n) >_{\mathsf{spo}} t}\ \text{SPO 1}$$

The induction hypothesis yields $s_i \geqslant_{\mathsf{wpo}} t$ for some i. By Lemma 6 and the subterm property of $\geqslant_{\mathcal{A}}$ we have $s \geqslant_{\mathcal{A}} t$. Thus, we obtain the following derivation of $s >_{\mathsf{wpo}} t$:

$$\frac{s \geqslant_{\mathcal{A}} t \qquad s_i \geqslant_{\mathsf{wpo}} t}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} t}\ \text{WPO 2a}$$

– Suppose that $s >_{\mathsf{spo}} t$ is derived as follows:

$$\frac{s \sqsupset t \qquad \forall j.\ \ s >_{\mathsf{spo}} t_j}{s = f(s_1, \ldots, s_n) >_{\mathsf{spo}} g(t_1, \ldots, t_n) = t}\ \text{SPO 2a}$$

According to the definition of $s \sqsupset t$, we further distinguish two subcases. If $s >_{\mathcal{A}} t$ then $s >_{\mathsf{wpo}} t$ is immediate. Otherwise, $s \geqslant_{\mathcal{A}} t$ and $f \succ g$ hold. In this case we derive $s >_{\mathsf{wpo}} t$ as follows:

$$\frac{s \geqslant_{\mathcal{A}} t \qquad f \succ g \qquad \dfrac{\forall j.\ \ s >_{\mathsf{spo}} t_j}{\forall j.\ \ s >_{\mathsf{wpo}} t_j}\ \text{I.H.}}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} g(t_1, \ldots, t_m) = t}\ \text{WPO 2b(i)}$$

– Suppose that $s >_{\mathsf{spo}} t$ is derived as follows:

$$\frac{s \sqsupseteq t \qquad \forall j.\ \ s >_{\mathsf{spo}} t_j \qquad (s_1, \ldots, s_n) >_{\mathsf{spo}}^{\mathsf{lex}} (t_1, \ldots, t_m)}{s = f(s_1, \ldots, s_n) >_{\mathsf{spo}} g(t_1, \ldots, t_m) = t}\ \text{SPO 2b}$$

Because of $s \sqsupseteq t$, we have $s >_{\mathcal{A}} t$ or both $s \geqslant_{\mathcal{A}} t$ and $f \succsim g$. In the former case $s >_{\mathsf{wpo}} t$ is immediate. In the latter case $s >_{\mathsf{wpo}} t$ is derived by WPO 2b(ii) as follows:

$$\frac{s \geqslant_{\mathcal{A}} t \qquad f \succsim g \qquad \dfrac{\forall j.\ \ s >_{\mathsf{spo}} t_j}{\forall j.\ \ s >_{\mathsf{wpo}} t_j}\ \text{I.H.} \qquad \dfrac{(s_1, \ldots, s_n) >_{\mathsf{spo}}^{\mathsf{lex}} (t_1, \ldots, t_m)}{(s_1, \ldots, s_n) >_{\mathsf{wpo}}^{\mathsf{lex}} (t_1, \ldots, t_m)}\ \text{I.H.}}{s = f(s_1, \ldots, s_n) >_{\mathsf{wpo}} g(t_1, \ldots, t_m) = t}$$

In any case we have $s >_{\mathsf{wpo}} t$. □

As a consequence, $>_{\mathsf{wpo}}$ and $>_{\mathsf{spo}}$ coincide, provided that $\mathcal{A}$ is simple. This result can be extended to monotonic semantic path orders.

Lemma 8. *The triple $(\geqslant_{\mathcal{A}}, \sqsupseteq, \sqsupset)$ is a reduction triple.* □

Let $>_{\mathsf{mspo}}$ denote the monotonic semantic path order induced from $\geqslant_{\mathcal{A}}$ and $>_{\mathsf{spo}}$. Since $s >_{\mathsf{wpo}} t$ implies $s \geqslant_{\mathcal{A}} t$ (Lemma 6), $s >_{\mathsf{mspo}} t$ is equivalent to $s >_{\mathsf{spo}} t$. By using this equivalence together with Lemmata 5 and 7, we obtain the following result.

Theorem 4. *The three orders $>_{\mathsf{wpo}}$, $>_{\mathsf{spo}}$, and $>_{\mathsf{mspo}}$ coincide, provided that $\mathcal{A}$ is simple.* □

5 Generalized Weighted Path Orders

According to Theorem 4, weighted path orders can be defined as monotonic semantic path orders. Moreover, Lemma 8 reveals that even for non-simple algebras the construction of reduction triples is valid. This observation suggests a generalization of weighted path orders, which does not impose simplicity on algebras. Besides, we exploit the fact that stable order pairs need not be closed under contexts, marking root symbols of function applications; see [1] and [5, Definition 5].

Let $\mathcal{F}$ be a signature. For each $f \in \mathcal{F}$ we associate a marked function symbol $f^\sharp \notin \mathcal{F}$ of the same arity. The set $\{f^\sharp \mid f \in \mathcal{F}\}$ is denoted by $\mathcal{F}^\sharp$. For each term $t = f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ we denote $f^\sharp(t_1, \ldots, t_n)$ by $t^\sharp$. Let $\mathcal{A}$ be a weakly monotone well-founded $(\mathcal{F} \cup \mathcal{F}^\sharp)$-algebra and $\succsim$ a well-founded precedence on $\mathcal{F}$. The pair $(\sqsupseteq^\sharp, \sqsupset^\sharp)$ of relations on $\mathcal{T}(\mathcal{F}, \mathcal{V}) \setminus \mathcal{V}$ is defined as follows: Let $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$. We write $s \sqsupseteq^\sharp t$ if $s^\sharp >_\mathcal{A} t^\sharp$, or $s^\sharp \geqslant_\mathcal{A} t^\sharp$ and $f \succsim g$. Similarly, we write $s \sqsupset^\sharp t$ if $s^\sharp >_\mathcal{A} t^\sharp$, or $s^\sharp \geqslant_\mathcal{A} t^\sharp$ and $f \succ g$. The relation $\gtrsim$ is defined as the restriction of $\geqslant_\mathcal{A}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

Proposition 2. *The triple $(\gtrsim, \sqsupseteq^\sharp, \sqsupset^\sharp)$ is a reduction triple on $\mathcal{T}(\mathcal{F}, \mathcal{V})$.* □

Definition 5. *The* generalized weighted path order (GWPO) $>_{\mathsf{gwpo}}$ *induced from $\mathcal{A}$ and $\succsim$ is the monotonic semantic path order induced from $(\gtrsim, \sqsupseteq^\sharp, \sqsupset^\sharp)$.*

Corollary 1. *Every generalized weighted path order is a reduction order, provided that the signature is bounded.* □

For convenience, we reformulate the definition of $>_{\mathsf{gwpo}}$ in the style of Definition 1.

Definition 6. *The relation $>_{\mathsf{wpo}'}$ is defined on terms as follows: $s >_{\mathsf{wpo}'} t$ if $s = f(s_1, \ldots, s_m)$ and one of the following conditions hold.*

1. *$s_i \geqslant_{\mathsf{wpo}'} t$ for some $1 \leqslant i \leqslant m$.*
2. *$t = g(t_1, \ldots, t_n)$, $s^\sharp \geqslant_\mathcal{A} t^\sharp$, and $s >_{\mathsf{wpo}'} t_j$ for all $1 \leqslant j \leqslant n$, and moreover*
   - *a. $s^\sharp >_\mathcal{A} t^\sharp$,*
   - *b. $f \succ g$, or*
   - *c. $f \succsim g$ and $(s_1, \ldots, s_m) >_{\mathsf{wpo}'}^{\mathsf{lex}} (t_1, \ldots, t_n)$.*

Proposition 3. *The SPO $>_{\mathsf{spo}}$ induced from $(\sqsupseteq^\sharp, \sqsupset^\sharp)$ coincides with $>_{\mathsf{wpo}'}$. For all terms s and t the relation $s >_{\mathsf{gwpo}} t$ is equivalent to $s \geqslant_\mathcal{A} t$ and $s >_{\mathsf{wpo}'} t$.* □

Corollary 2. *The relations $>_{\mathsf{gwpo}}$ and $>_{\mathsf{wpo}}$ coincide, provided that $\mathcal{A}$ is simple and $f_\mathcal{A}(x_1, \ldots, x_n) = f^\sharp_\mathcal{A}(x_1, \ldots, x_n)$ for all $f^{(n)} \in \mathcal{F}$.* □

Since polynomial interpretation orders [15] and Knuth–Bendix orders [13] as well as LPOs are simulated by WPOs [27], they are also subsumed by GWPOs. We demonstrate termination proofs by GWPOs with a few examples. All examples are not handled by WPOs.

Example 7. Consider the TRS $\mathcal{R}$ for round-up division:

$$\begin{array}{lll} \mathsf{p}(\mathsf{0}) \to \mathsf{0} & x - \mathsf{0} \to x & \mathsf{0} \div \mathsf{s}(y) \to \mathsf{0} \\ \mathsf{p}(\mathsf{s}(x)) \to x & x - \mathsf{s}(y) \to \mathsf{p}(x) - y & \mathsf{s}(x) \div \mathsf{s}(y) \to \mathsf{s}((x - y) \div \mathsf{s}(y)) \end{array}$$

Let $\mathcal{A}$ be the weakly monotone algebra on $\mathbb{N}$ with the interpretations

$$\begin{array}{lllll} \mathsf{0}_{\mathcal{A}} = 0 & \mathsf{s}_{\mathcal{A}}(x) = x + 1 & \mathsf{p}_{\mathcal{A}}(x) = x & x -_{\mathcal{A}} y = x & x \div_{\mathcal{A}} y = x \\ \mathsf{0}^{\sharp}_{\mathcal{A}} = 0 & \mathsf{s}^{\sharp}_{\mathcal{A}}(x) = 0 & \mathsf{p}^{\sharp}_{\mathcal{A}}(x) = 0 & x -^{\sharp}_{\mathcal{A}} y = y & x \div^{\sharp}_{\mathcal{A}} y = x + y \end{array}$$

and let $\gtrsim$ be an arbitrary precedence. The GWPO induced from $\mathcal{A}$ and $\gtrsim$ orients all rules in $\mathcal{R}$. In particular, $x - \mathsf{s}(y) >_{\mathsf{wpo}'} \mathsf{p}(x) - y$ is derived from the inequalities $x -^{\sharp} \mathsf{s}(y) >_{\mathcal{A}} \mathsf{p}(x) -^{\sharp} y$ and $x -^{\sharp} \mathsf{s}(y) >_{\mathcal{A}} \mathsf{p}^{\sharp}(x)$.

Example 8. Consider the TRS $\mathcal{R}$ taken from [2, Example 4.28], which computes the bit length of a natural number:

$$\begin{array}{lll} \mathsf{half}(\mathsf{0}) \to \mathsf{0} & \mathsf{half}(\mathsf{s}(\mathsf{0})) \to \mathsf{0} & \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{s}(\mathsf{half}(x)) \\ \mathsf{bits}(\mathsf{0}) \to \mathsf{0} & \mathsf{bits}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x)))) & \end{array}$$

Let $\mathcal{A}$ be the weakly monotone algebra on $\mathbb{N}$ with:

$$\begin{array}{llll} \mathsf{0}_{\mathcal{A}} = 0 & \mathsf{s}_{\mathcal{A}}(x) = x + 1 & \mathsf{half}_{\mathcal{A}}(x) = \max\{0, x - 1\} & \mathsf{bits}_{\mathcal{A}}(x) = x \\ \mathsf{0}^{\sharp}_{\mathcal{A}} = 0 & \mathsf{s}^{\sharp}_{\mathcal{A}}(x) = x + 1 & \mathsf{half}^{\sharp}_{\mathcal{A}}(x) = \max\{0, x - 1\} & \mathsf{bits}^{\sharp}_{\mathcal{A}}(x) = x \end{array}$$

The GWPO $>_{\mathsf{gwpo}}$ induced by $\mathcal{A}$ and a precedence $\gtrsim$ with $\mathsf{half}, \mathsf{bits} \succ \mathsf{s}$ satisfies $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$ as $\ell \geqslant_{\mathcal{A}} r$ and $\ell >_{\mathsf{wpo}'} r$ for all rules $\ell \to r \in \mathcal{R}$. In particular, $\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x))))$ is derived as follows. The inequality $\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{bits}(\mathsf{half}(\mathsf{s}(x)))$ is derived from repeated application of WPO′ 2a:

$$\dfrac{\mathsf{bits}^{\sharp}(\mathsf{s}(x)) >_{\mathcal{A}} \mathsf{bits}^{\sharp}(\mathsf{half}(\mathsf{s}(x))) \qquad \dfrac{\mathsf{bits}^{\sharp}(\mathsf{s}(x)) >_{\mathcal{A}} \mathsf{half}^{\sharp}(\mathsf{s}(x)) \qquad \dfrac{\mathsf{s}(x) \geqslant_{\mathsf{wpo}'} \mathsf{s}(x)}{\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{s}(x)}}{\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{half}(\mathsf{s}(x))}}{\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{bits}(\mathsf{half}(\mathsf{s}(x)))}$$

Thus, $\mathsf{bits}(\mathsf{s}(x)) >_{\mathsf{wpo}'} \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x))))$ follows from WPO′ 2b with $\mathsf{bits} \succ \mathsf{s}$ and $\mathsf{bits}^{\sharp}(\mathsf{s}(x)) \geqslant_{\mathcal{A}} \mathsf{s}^{\sharp}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x))))$.

6 Experimental Results

In order to evaluate GWPOs in termination analysis we implemented a prototype termination tool based on Proposition 1 and Corollary 1. Following the automation techniques of WPO [27], we search a suitable weakly monotone well-founded algebra from two classes of algebras over $\mathbb{N}$: One is *linear interpretation* and the other is *max/plus interpretation*. Since simplicity of algebras is not required for GWPOs, we may use more general forms of interpretations.

Linear Interpretations. Algebras $\mathcal{A}$ of this class use linear polynomials over $\mathbb{N}$ like Example 7. For each $f^{(n)} \in \mathcal{F} \cup \mathcal{F}^\sharp$ its interpretation is of the form $f_\mathcal{A}(x_1, \ldots, x_n) = c_0 + c_1 x_1 + \cdots + c_n x_n$ where $c_0 \in \mathbb{N}$ and $c_1, \ldots, c_n \in \{0, 1\}$. Simple monotone algebras for WPOs[3] are obtained by setting $c_1 = \cdots = c_n = 1, f_\mathcal{A} = f^\sharp_\mathcal{A}$ for all $f^{(n)} \in \mathcal{F}$, and those for Knuth–Bendix orders (KBOs) are obtained by further restriction for admissibility, see [27]. Comparison of linear polynomials is reduced to that of coefficients by using the following trivial fact:

Proposition 4. *Let $f(x_1, \ldots, x_n) = c_0 + c_1 x_1 + \cdots + c_n x_n$ and $g(x_1, \ldots, x_n) = d_0 + d_1 x_1 + \cdots + d_n x_n$ be linear polynomials over $\mathbb{N}$. The next statements hold.*

- *$f \geqslant g$ if and only if $c_0 \geqslant d_0$ and $c_i \geqslant d_i$ for all $1 \leqslant i \leqslant n$.*
- *$f > g$ if and only if $c_0 > d_0$ and $c_i \geqslant d_i$ for all $1 \leqslant i \leqslant n$.*

Here $f \geqslant g$ ($f > g$) means that $f(a_1, \ldots, a_n) \geqslant g(a_1, \ldots, a_n)$ ($f(a_1, \ldots, a_n) > g(a_1, \ldots, a_n)$) for all $a_1, \ldots, a_n \in \mathbb{N}$.

Max/plus Interpretations. Algebras $\mathcal{A}$ of this class use a combination of $+$ and max like Example 8. For each $f^{(n)} \in \mathcal{F} \cup \mathcal{F}^\sharp$ its interpretation is of the form $f_\mathcal{A}(x_1, \ldots, x_n) = \max\{c_0, c_1 + c'_1 x_1, \cdots, c_n + c'_n x_n\}$ where $c_0 \in \mathbb{N}, c_1, \ldots, c_n \in \mathbb{Z}$ and $c'_1, \ldots, c'_n \in \{0, 1\}$. Simple monotone algebras for WPOs are obtained by imposing $c_1, \ldots, c_n \in \mathbb{N}, c'_1 = \cdots = c'_n = 1, f_\mathcal{A} = f^\sharp_\mathcal{A}$ for all $f^{(n)} \in \mathcal{F}$, and algebras for lexicographic path orders (LPOs) are obtained by additionally setting $c_0 = c_1 = \cdots = c_n = 0$ for all $f^{(n)} \in \mathcal{F}$ as in [27]. The restriction $c_1, \ldots, c_n \in \mathbb{N}$ is necessary for WPOs because allowing $c_1, \ldots, c_n < 0$ results in non-simple interpretations such as $\max\{0, x - 1\}$. Under this form of algebras, an interpretation of a term is flattened to the form of $\max\{g_1, \ldots, g_m\}$ where $g_1, \ldots, g_m$ are linear polynomials over $\mathbb{N}$. So comparison of max/plus interpretation is reduced to that of coefficients, using the following trivial fact and Proposition 4 in turn:

Proposition 5. *Let G and H be non-empty sets of linear polynomials over $\mathbb{N}$. The next statements hold.*

- *$\max G \geqslant \max H$ if and only if for every $h \in H$ there exists a linear polynomial $g \in G$ with $g \geqslant h$.*
- *$\max G > \max H$ if and only if for every $h \in H$ there exists a linear polynomial $g \in G$ with $g > h$.*

Since precedence constraints can be regarded as inequalities on natural numbers [28], searching a suitable combination of a precedence and an interpretation is done by solving linear arithmetic constraints (with if-then-else expressions).

The problem set for experiments consists of 1511 term rewrite systems from version 11.3 of the Termination Problem Database (TPDB) [21]. The reference implementation uses the SMT solver Z3 [17] as an external tool for solving linear constraints. The experiments were run on a PC with Intel Core i7-1065G7 CPU (1.30 GHz) and 16 GB memory.

---

[3] Our WPOs based on linear interpretations correspond to WPO($\mathcal{S}um$) by Yamada et al. [27] but without status functions.

Table 1. Experiments on 1511 TRSs from TPDB 11.3.

interpretations	*linear*			*max/plus*		
order	KBO	WPO	GWPO	LPO	WPO	GWPO
proved TRSs	103	122	357	149	221	385
timeouts (60 sec)	*8*	*9*	*9*	*12*	*12*	*28*

Now let us discuss the experimental results.[4] Table 1 shows that, as a whole, use of non-simple algebras substantially improves termination analysis, at the small cost of extra running time. In particular, in the case of linear interpretation, GWPOs significantly outperform WPOs. As a matter of fact, linear WPOs are unable to orient variable duplicating rules $\ell \to r$ such as $\mathsf{f}(x) \to \mathsf{g}(x, x)$ since $\ell \geqslant_{\mathcal{A}} r$ cannot be satisfied, but this does not apply to GWPOs based on linear interpretations with $\{0, 1\}$-coefficients. In the case of max/plus interpretations there are two TRSs (with over 100 rules) that are proved to be terminating by WPOs, but not by GWPOs due to the time limit. This indicates that using non-simple algebras for max/plus interpretation can result in increase of search space. This is not the case for linear interpretations.

7 Simulating Dependency Pairs by GWPOs

The powerfulness of GWPOs revealed in Sect. 6 can partly be explained by the fact that GWPO is capable of simulating a basic result of the dependency pair method [1]. To show the fact, we recall the dependency pair method. The set $\mathsf{DP}(\mathcal{R})$ of *dependency pairs* of a TRS $\mathcal{R}$ is defined as follows:

$$\mathsf{DP}(\mathcal{R}) = \{\ell^\sharp \to g^\sharp(t_1, \ldots, t_n) \mid \ell \to r \in \mathcal{R},\ r \trianglerighteq g(t_1, \ldots, t_n), \text{ and } g \in \mathcal{D}_{\mathcal{R}}\}$$

An order pair $(\geqslant, \sqsupset)$ on terms is a *reduction pair* if $\geqslant$ is a rewrite preorder and $\sqsupset$ is a well-founded stable order. The following theorem states a basic result of the dependency pair method.

Theorem 5 ([1]). *A TRS $\mathcal{R}$ is terminating if $\mathcal{R} \subseteq \geqslant$ and $\mathsf{DP}(\mathcal{R}) \subseteq \sqsupset$ for some reduction pair $(\geqslant, \sqsupset)$.*

We illustrate Theorem 5, using the fact that every weakly monotone algebra $\mathcal{A}$ on $\mathbb{N}$ induces the reduction pair $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$.

Example 9. Consider the TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{f}(x))), \mathsf{f}(x) \to \mathsf{g}(x)\}$. We show the termination of $\mathcal{R}$ using Theorem 5. The set $\mathsf{DP}(\mathcal{R})$ consists of the two dependency pairs:

$$\mathsf{f}^\sharp(\mathsf{f}(x)) \to \mathsf{f}^\sharp(\mathsf{g}(\mathsf{f}(x))) \qquad\qquad \mathsf{f}^\sharp(\mathsf{f}(x)) \to \mathsf{f}^\sharp(x)$$

[4] The implementation and the detailed experimental data are available at: https://www.jaist.ac.jp/project/maxcomp/23frocos/

By taking the $\{\mathsf{f}, \mathsf{g}, \mathsf{f}^\sharp, \mathsf{g}^\sharp\}$-algebra $\mathcal{A}$ with the interpretations

$$\mathsf{f}_\mathcal{A}(x) = x + 1 \qquad \mathsf{g}_\mathcal{A}(x) = 0 \qquad \mathsf{f}^\sharp_\mathcal{A}(x) = x \qquad \mathsf{g}^\sharp_\mathcal{A}(x) = 1$$

the inclusions $\mathcal{R} \subseteq \geqslant_\mathcal{A}$ and $\mathsf{DP}(\mathcal{R}) \subseteq >_\mathcal{A}$ hold. Hence, $\mathcal{R}$ is terminating.

We show that every termination proof by Theorem 5 with a weakly monotone algebra on $\mathbb{N}$ can be simulated by a GWPO. This class of algebras include linear polynomial interpretations and max/plus interpretations described in Sect. 6. Let $\mathcal{R}$ be a TRS and $\mathcal{A}$ a weakly monotone $(\mathcal{F} \cup \mathcal{F}^\sharp)$-algebra on $\mathbb{N}$ satisfying $\mathcal{R} \subseteq \geqslant_\mathcal{A}$ and $\mathsf{DP}(\mathcal{R}) \subseteq >_\mathcal{A}$. Define the $(\mathcal{F} \cup \mathcal{F}^\sharp)$-algebra $\mathcal{B}$ on $\mathbb{N}$ by

$$\begin{aligned} f_\mathcal{B}(a_1, \ldots, a_n) &= f_\mathcal{A}(a_1, \ldots, a_n) \\ f^\sharp_\mathcal{B}(a_1, \ldots, a_n) &= \begin{cases} f^\sharp_\mathcal{A}(a_1, \ldots, a_n) + 1 & \text{if } f \in \mathcal{D}_\mathcal{R} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

for each $f^{(n)} \in \mathcal{F}$. Let $>_{\mathsf{gwpo}}$ and $>_{\mathsf{wpo}'}$ denote the orders induced from $\mathcal{B}$ and an arbitrary but fixed precedence. First, let us see that $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$ holds for the last example.

Example 10 (continued from Example 9). The corresponding algebra $\mathcal{B}$ is:

$$\mathsf{f}_\mathcal{B}(x) = x + 1 \qquad \mathsf{g}_\mathcal{B}(x) = 0 \qquad \mathsf{f}^\sharp_\mathcal{B}(x) = x + 1 \qquad \mathsf{g}^\sharp_\mathcal{B}(x) = 0$$

We have $\mathcal{R} \subseteq \geqslant_\mathcal{B}$ by construction. The inequality $\mathsf{f}(\mathsf{f}(x)) >_{\mathsf{wpo}'} \mathsf{f}(\mathsf{g}(\mathsf{f}(x)))$ is derived by successive application of WPO′ 2a as follows:

$$\dfrac{\mathsf{f}^\sharp(\mathsf{f}(x)) >_\mathcal{B} \mathsf{f}^\sharp(\mathsf{g}(\mathsf{f}(x))) \qquad \dfrac{\mathsf{f}^\sharp(\mathsf{f}(x)) >_\mathcal{B} \mathsf{g}^\sharp(\mathsf{f}(x)) \qquad \dfrac{\mathsf{f}^\sharp(\mathsf{f}(x)) >_\mathcal{B} \mathsf{f}^\sharp(x) \qquad \dfrac{x \geqslant_{\mathsf{wpo}'} x}{\mathsf{f}(x) >_{\mathsf{wpo}'} x}}{\mathsf{f}(\mathsf{f}(x)) >_{\mathsf{wpo}'} \mathsf{f}(x)}}{\mathsf{f}(\mathsf{f}(x)) >_{\mathsf{wpo}'} \mathsf{g}(\mathsf{f}(x))}}{\mathsf{f}(\mathsf{f}(x)) >_{\mathsf{wpo}'} \mathsf{f}(\mathsf{g}(\mathsf{f}(x)))}$$

The inequality $\mathsf{f}(x) >_{\mathsf{wpo}'} \mathsf{g}(x)$ follows from $\mathsf{f}^\sharp(x) >_\mathcal{B} \mathsf{g}^\sharp(x)$. Hence $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$. Note that neither $\mathsf{f}^\sharp(\mathsf{f}(x)) >_\mathcal{A} \mathsf{g}^\sharp(\mathsf{f}(x))$ nor $\mathsf{f}^\sharp(x) >_\mathcal{A} \mathsf{g}^\sharp(x)$ holds.

Now we verify that $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$ holds in general. By construction $\mathcal{R} \subseteq \geqslant_\mathcal{B}$ is immediate from $\mathcal{R} \subseteq \geqslant_\mathcal{A}$. So it remains to show $\mathcal{R} \subseteq >_{\mathsf{wpo}'}$. We prove the following stronger property.

Lemma 9. *Let $\ell \to r \in \mathcal{R}$. For every subterm t of r the relation $\ell >_{\mathsf{wpo}'} t$ holds.*

Proof. We use structural induction on t. If t is a variable, then x must be a subterm of ℓ, and thus $\ell >_{\mathsf{wpo}'} t$. Otherwise, t is in the form of $g(t_1, \ldots, t_n)$. The induction hypothesis yields $\ell >_{\mathsf{wpo}'} t_j$ for all $1 \leqslant j \leqslant n$. We claim $\ell^\sharp >_\mathcal{B} t^\sharp$, from which the desired inequality $\ell >_{\mathsf{wpo}'} t$ follows by WPO′ 2a. To show the claim, consider an arbitrary assignment α for $\mathcal{B}$. Depending on g, we distinguish two cases.

– If $g \notin \mathcal{D}_\mathcal{R}$ then $[\alpha]_\mathcal{B}(\ell^\sharp) = [\alpha]_\mathcal{A}(\ell^\sharp) + 1 > 0 = [\alpha]_\mathcal{B}(t^\sharp)$.
– If $g \in \mathcal{D}_\mathcal{R}$ then $\ell^\sharp \to t^\sharp \in \mathsf{DP}(\mathcal{R})$. The assumption $\mathsf{DP}(\mathcal{R}) \subseteq >_\mathcal{A}$ yields the inequality $\ell^\sharp >_\mathcal{A} t^\sharp$. Thus, $[\alpha]_\mathcal{B}(\ell^\sharp) = [\alpha]_\mathcal{A}(\ell^\sharp) + 1 > [\alpha]_\mathcal{A}(t^\sharp) + 1 = [\alpha]_\mathcal{B}(t^\sharp)$.

In either case $[\alpha]_\mathcal{B}(\ell^\sharp) > [\alpha]_\mathcal{B}(t^\sharp)$ is obtained. Hence, $\ell^\sharp >_\mathcal{B} t^\sharp$ holds. □

Theorem 6. *The inclusion $\mathcal{R} \subseteq >_{\mathsf{gwpo}}$ holds.* □

8 Conclusion

We have shown that weighted path orders can be simulated by a suitable variant of SPOs based on order pairs, and introduced a generalization of WPOs whose termination proving power goes beyond the realm of simple termination. To conclude the paper, we discuss related work and future work.

Simulating KBOs by SPOs. A key observation for simulating WPOs by SPOs is that weight comparison can be simulated by successive application of SPO 1 and SPO 2a as observed in Example 6. Another observation is that the SPOs are already reduction orders without a help of harmonious rewrite preorders. These two observations owe to Geser's work [9, Theorem 5], where it is shown that extended KBOs [7, Sect. 5] can be simulated by SPOs. Unifying our result and Geser's result is future work.

General Path Orders. In this paper the lexicographic versions of path orders were investigated. However, it is very likely that the same result can be obtained even if we adopt multiset comparison or status functions. General path orders (GPOs) [8,10] are a unifying framework for such extensions, parameterizing the way to compare arguments. It is worth investigating simulation results between GPOs and WPOs by extending the parameters of GPOs so as to take order pairs.

Reduction Pairs Based on WPOs. In order to build reduction pairs from WPOs Yamada et al. [27, Sect. 4] extended the definition of WPOs by the notion of *partial status function* π. The extension allows us to specify argument positions $\pi(f) = [i_1, \ldots, i_m]$ compared in WPO 2b and WPO 2b(ii) for each function symbol $f^{(n)} \in \mathcal{F} \cup \mathcal{F}^\sharp$. We anticipate that partial status functions can also be integrated into GWPOs and the thus-obtained version characterizes the reduction pair version of WPOs.

Acknowledgements. We are grateful to Vincent van Oostrom for his valuable questions and comments on our preliminary work. We also thank Alfons Geser for his support on literature. The suggestions by the anonymous referees greatly helped to improve the presentation of the paper.

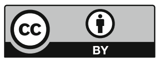
References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoret. Comput. Sci. **236**, 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
2. Arts, T., Giesl, J.: A collection of examples for termination of term rewriting using dependency pairs. Tech. rep, RWTH Aachen (2001)
3. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge Univ. Press (1998). https://doi.org/10.1017/CBO9781139172752
4. Borralleras, C.: Ordering-Based Methods for Proving Termination Automatically. Ph.D. thesis, Universitat Politècnica de Catalunya (2003)
5. Borralleras, C., Ferreira, M., Rubio, A.: Complete monotonic semantic path orderings. In: Proc. 17th International Conference on Automated Deduction. LNCS (LNAI), vol. 1831, pp. 346–364 (2000). https://doi.org/10.1007/10721959_27
6. Buchholz, W.: Proof-theoretic analysis of termination proofs. Ann. Pure Appl. Logic **75**, 57–65 (1995). https://doi.org/10.1016/0168-0072(94)00056-9
7. Dershowitz, N.: Orderings for term-rewriting systems. Theoret. Comput. Sci. **17**, 279–301 (1982). https://doi.org/10.1016/0304-3975(82)90026-3
8. Dershowitz, N., Hoot, C.: Natural termination. Theoret. Comput. Sci. **142**, 179–207 (1995). https://doi.org/10.1016/0304-3975(94)00275-4
9. Geser, A.: On a monotonic semantic path ordering. Tech. Rep. 92–13, Ulmer Informatik-Berichte, Universität Ulm, Germany (1992)
10. Geser, A.: An improved general path order. Appl. Algebra Eng. Commun. Comput. **7**, 469–511 (1996). https://doi.org/10.1007/BF01293264
11. Jakubův, J., Kaliszyk, C.: Relaxed weighted path order in theorem proving. Math. Comput. Sci. **14**(3), 657–670 (2020). https://doi.org/10.1007/s11786-020-00474-0
12. Kamin, S., Lévy, J.: Two generalizations of the recursive path ordering. Tech. rep., University of Illinois (1980), unpublished manuscript
13. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
14. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Proc. 20th International Conference on Rewriting Techniques and Applications. LNCS, vol. 5595, pp. 295–304 (2009). https://doi.org/10.1007/978-3-642-02348-4_21
15. Lankford, D.: On proving term rewriting systems are noetherian. Louisiana Technical University, Tech. rep. (1979)
16. Middeldorp, A., Zantema, H.: Simple termination of rewrite systems. Theoret. Comput. Sci. **175**, 127–158 (1997). https://doi.org/10.1016/S0304-3975(96)00172-7
17. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Saito, T., Hirokawa, N.: Toma 0.2: An equational theorem prover. In: Proceedings of 11th Workshop on Confluence. p. 59 (2022), the tool is available at https://www.jaist.ac.jp/project/maxcomp/
19. Sternagel, C., Thiemann, R.: Formalizing Knuth–Bendix orders and Knuth–Bendix completion. In: Proc. 13th International Conference on Rewriting Techniques and Applications. LIPIcs, vol. 21, pp. 287–302 (2013). https://doi.org/10.4230/LIPIcs.RTA.2013287

20. Terese: Term Rewriting Systems. Cambridge University Press (2003)
21. Termination Community: The Termination Problem Database (TPDB). https://github.com/TermCOMP/TPDB. Accessed 12 May 2023
22. Toyama, Y.: Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In: Proc. 15th International Conference on Rewriting Techniques and Applications. LNCS, vol. 3091, pp. 40–54 (2004). https://doi.org/10.1007/978-3-540-25979-4_3
23. Winkler, S., Moser, G.: MædMax: A Maximal Ordered Completion Tool. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Proc. 9th International Joint Conference on Automated Reasoning. LNCS (LNAI), vol. 10900, pp. 472–480. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_31
24. Yamada, A.: Towards a unified method for termination. In: Proc. 16th Workshop on Termination, pp. 248–267 (2018), the presentation slides are available at http://wst2018.webs.upv.es/
25. Yamada, A.: Term orderings for non-reachability of (conditional) rewriting. In: Proc. 11th International Joint Conference on Automated Reasoning. LNCS, vol. 13385, pp. 248–267 (2022). https://doi.org/10.1007/978-3-031-10769-6_15
26. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya Termination Tool. In: Proc. Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications. vol. 8560, pp. 466–475 (2014). https://doi.org/10.1007/978-3-319-08918-8_32
27. Yamada, A., Kusakari, K., Sakabe, T.: A unified ordering for termination proving. Sci. Comput. Program. **111**, 110–134 (2015). https://doi.org/10.1016/j.scico.2014.07.009
28. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. J. Autom. Reason. **43**, 173–201 (2009)

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.



KBO Constraint Solving Revisited

Yasmine Briefs[1,2](✉), Hendrik Leidinger[1,2], and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarbrücken, Germany
{ybriefs,hleiding,weidenbach}@mpi-inf.mpg.de
[2] Graduate School of Computer Science,
Saarland Informatics Campus, Saarbrücken, Germany

Abstract. KBO constraint solving is very well-known to be an NP-complete problem. Motivated by the needs of the family of SCL calculi, we consider the particular case where all terms occurring in a constraint are bound by a (single) ground term. We show that this problem and variants of this problem remain NP-complete even if the form of atoms in the constraint is further restricted. In addition, for a non-strict, partial term ordering solely based on symbol counting constraint solving remains NP-complete. Nevertheless, we provide a new simple algorithm testing KBO constraint solvability that performs well on benchmark examples.

Keywords: KBO Constraint Solving · NP-complete problem · Weight Ordering Constraint Solving

1 Introduction

The family of SCL calculi (Clause Learning from Simple Models) [2,5,13] perform reasoning on a set of first-order clauses. They develop a trail of ground literals with respect to a ground term (atom) bound β and an ordering $\prec$. All ground literals on the trail are $\prec$ (or $\preceq$) smaller than the ground term (atom) β and $\prec$ should in particular have the property that for any term t there are only finitely many literals s such that $s \prec t$. In case SCL does not detect a conflict with respect to a finite, exhaustive trail of ground literals, they constitute a model candidate for the clause set [4]. If SCL detects a conflict it learns a new first-order non-ground clause. It is derived by resolution and factoring with guidance from the trail. A natural choice for the ordering $\prec$ is the Knuth-Bendix (KBO) ordering [9]. For the ground case, a KBO relation can be efficiently computed [14]. All SCL calculi propagate literals from clauses with respect to the trail. For example, given a trail $[P(a)]$ and a clause $\neg P(x) \lor R(x, y)$ the literal $R(a, y)$ could be propagated. The SCL theory only enables ground literals on the trail, however, in practice it is not affordable to put all groundings of $R(a, y)$ on the trail that are $\prec$ smaller than β. Therefore, we already considered trail literals with variables when we developed a two-watched literal scheme for SCL [3]. Recall that this propagation situation is not exceptional as typically not all literals in a clause carry all occurring variables. The consequence of this

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 81–98, 2023.
https://doi.org/10.1007/978-3-031-43369-6_5

extension is that for SCL we now need to decide solvability of conjunctions of inequations $t_i \prec \beta$ where the t_i may contain (shared) variables, i.e., we have to decide solvability of a particular form of KBO constraints if $\prec$ is the KBO.

For the SCL(EQ) calculus [13] the requirements on constraint solving get more sophisticated. Now the trail is a sequence of unit (in)equalities and propagation and conflicting clauses are decided with respect to the resulting congruence. For an extended congruence closure algorithm [6,8,15,16] we need now in addition to inequations $t_i \prec \beta$ to consider inequalities $t_i \neq s_i$ in order to separate congruence classes. In its simplest form, constraints consist of inequations $t_i \prec \beta$ and inequalities $t_i \neq s_i$ where β and the s_i are ground, so called *simple right-ground* constraints, Definition 4. In a more general setting, the s_i carry variables and then a quantifier alternation on variables occurring in s_i but not in t_i needs to be considered. Such constraints are called *alternating*, Definition 27.

In this paper we investigate the complexity of all these variants with respect to a KBO $<$, Definitions 3, 4, 25, 27, but also a weaker non-strict ordering based on pure symbol counting, Definition 22. Except for constraints bound by a single ground term, Proposition 26, all problems are NP-hard, Propositions 5, 21, 24, 28.

Korovin and Voronkov developed a decision procedure [10] for KBO constraints consisting of inequations $s_j < t_j$ only and refined it to an NP algorithm [11]. According to Löchner [14], these results are "of more theoretical interest" because they are "too involved to be implemented with reasonable effort". In fact, to the best of our knowledge we present the first implemented algorithm for KBO constraint solving in this paper. Later, Korovin and Voronkov [12] showed that checking satisfiability of a KBO constraint consisting of a single inequation $s < t$ can be done in polynomial time. For the special case of a right-ground constraint consisting of a single inequation $s < t$, what their algorithm essentially does is assigning the minimal constant to every variable.

To the best of our knowledge the problem of simple right-ground KBO constraints has never been studied before. We are also not aware of any implementation of a KBO constraint solving algorithm. The paper is now organized as follows: In Sect. 3 we prove the NP-completeness of this problem and present an algorithm to solve it. In Sect. 4 we study the complexity of variants of this problem including alternating constraints. We also consider a non-strict, partial ordering based on symbol counting and weaker than a KBO. The algorithm for right-ground constraints is extended to alternating constraints. In Sect. 5 we put the algorithm developed in Sects. 3 and 4 to practice and end the paper with a discussion of the obtained results, Sect. 6.

2 Preliminaries

In the following let Σ be a *signature*, i.e., a finite set of function symbols. Every function symbol f has an associated *arity* which we denote by $\text{arity}(f)$. Function symbols c with $\text{arity}(c) = 0$ are called *constants*. We denote the set of all *terms* by $T(\Sigma, \mathcal{X})$ where $\mathcal{X}$ is an infinite set of variables. $\mathit{Vars}(t)$ denotes the set of

variables occurring in the term t. A term t is called *ground* if it contains no variables, i.e., $Vars(t) = \emptyset$. The set of all *ground terms* is denoted by $T(\Sigma)$. We assume that Σ contains at least one non-constant function and at least one constant, i.e., that $T(\Sigma)$ is infinite. For otherwise, constraint solving becomes trivial. A *substitution* is a mapping $\sigma : \mathcal{X} \to T(\Sigma, \mathcal{X})$ such that $\sigma(x) \neq x$ for only finitely many $x \in \mathcal{X}$. The application $t\sigma$ of a substitution σ to a term $t \in T(\Sigma, \mathcal{X})$ is defined in the usual way. We call a substitution *grounding* for some term $t \in T(\Sigma, \mathcal{X})$ if $t\sigma$ is ground. A substitution σ is a *matcher* from s to t if $s\sigma = t$. We consider the following version of the Knuth-Bendix ordering (KBO) on ground terms:

Definition 1 (KBO on Ground Terms [9]). *Let $\succ$ be a strict total ordering (a precedence) on Σ, and $w : \Sigma \to \mathbb{N}^+$ a weight function. w is extended to terms recursively by $w(f(t_1, \dots, t_n)) = w(f) + \sum_{i=1}^{n} w(t_i)$. The Knuth-Bendix ordering $>_{KBO}$ induced by $\succ$ and w is defined by $s >_{KBO} t$ iff*

1. *$w(s) > w(t)$, or*
2. *$w(s) = w(t)$, and*
   (a) $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$ and $f \succ g$, or
   (b) $s = f(s_1, \dots, s_m)$, $t = f(t_1, \dots, t_m)$ and $(s_1, \dots, s_m) >_{KBO}^{lex} (t_1, \dots, t_m)$.

In particular, the precedence is strict and total, no unary function f with $w(f) = 0$ is allowed and all weights are natural numbers. It can be shown that $>_{\mathrm{KBO}}$ is a strict, total and well-founded ordering on ground terms. In the following, we simply write $>$ for $>_{\mathrm{KBO}}$.

Definition 2. *A KBO constraint C is a finite set of atoms $t\#s$ where $t, s \in T(\Sigma, \mathcal{X})$ and $\# \in \{<, >, \neq, \leq, \geq, =\}$. We say that $C = \{t_1 \#_1 s_1, \dots, t_n \#_n s_n\}$ is satisfiable if there exists a substitution σ that is grounding for all t_j, s_j such that*

$$\bigwedge_{j=1}^{n} t_j\sigma \;\#_j\; s_j\sigma.$$

Such a grounding substitution σ is called a solution.

Definition 3. *A right-ground KBO constraint C is a KBO constraint where $s_1, \dots, s_n \in T(\Sigma)$, i.e., only the t_j may contain variables.*

Definition 4. *A simple right-ground KBO constraint C is a right-ground KBO constraint where $\# \in \{<, \neq\}$.*

For simple right-ground KBO constraints, we prefer more explicit notation: We now assume $t_1, \dots, t_n, l_1, \dots, l_m \in T(\Sigma, \mathcal{X})$, $s_1, \dots, s_n, r_1, \dots, r_m \in T(\Sigma)$ and call C satisfiable if there exists a substitution σ that is grounding for all t_j, l_j such that

$$\left(\bigwedge_{j=1}^{n} t_j\sigma < s_j\right) \wedge \left(\bigwedge_{j=1}^{m} l_j\sigma \neq r_j\right).$$

3 Simple, Right-Ground KBO Constraints

We start by investigating the complexity of simple, right-ground KBO constraint solving.

Proposition 5. *Checking satisfiability for simple right-ground KBO constraints is NP-hard.*

Proof. We reduce from MONOTONE 3SAT which is NP-complete by [7]. Let $N \uplus M$ be a set of clauses where N consists of the clauses with only positive literals and M consists of the clauses with only negative literals. We consider a signature with a constant a, a ternary function f and a unary function g. We use a KBO instance where all weights are 1 and $f \succ g \succ a$. For every propositional variable P occurring in $N \uplus M$, we introduce a variable x_P. Then the equation $x_P = a$ stands for P is *true* and $x_P \neq a$ stands for P is *false*.

Now every positive clause $(P \vee Q \vee R) \in N$ is encoded as an inequation $f(x_P, x_Q, x_R) < f(g(a), g(a), g(a))$. Obviously, this inequation can only be satisfied by a grounding that maps at least one of these variables to a, i.e., that sets at least one of P, Q, R to *true*.

Every negative clause $(\neg P \vee \neg Q \vee \neg R) \in M$ is encoded as an inequality $f(x_P, x_Q, x_R) \neq f(a, a, a)$. Obviously, this can only be satisfied if not all of these variables are mapped to a, i.e., if at least one of P, Q, R is *false*.

Now the clause set has a solution iff there is a solution to the constructed simple right-ground KBO constraint. Assume $N \uplus M$ is satisfiable by a valuation β. Then for every propositional variable P map x_P to a if $\beta(P) = 1$ and to $g(a)$ otherwise. As explained above, this grounding will satisfy the constraint. Now let σ be a solution to the constraint. Then the valuation β where $\beta(P) = 1$ if $\sigma(x_P) = a$ and $\beta(P) = 0$ otherwise satisfies $N \uplus M$.

We have added $|M|$ inequalities and $|N|$ inequations which can be constructed in polynomial time, so the reduction works in polynomial time. □

Proposition 6. *Checking satisfiability for simple right-ground KBO constraints is in NP.*

Proof. Let $C = \{t_1 < s_1, \ldots, t_n < s_n, l_1 \neq r_1, \ldots, l_m \neq r_m\}$ be a constraint. If for some inequality $l_j \neq r_j$, there is no matcher from l_j to r_j, we can ignore this inequality since it is true for every grounding. If for some inequality $l_j \neq r_j$, it actually holds that $l_j = r_j$, then this inequality is impossible to satisfy, so we are done. After sorting out these two cases, as r_j is ground, every inequality $l_j \neq r_j$ has a unique matcher τ_j which has linear size with respect to r_j. In the following, we say that the term $\tau_j(x)$ is restricted by the inequality $l_j \neq r_j$. The inequality $l_j \neq r_j$ then signifies

$$\bigvee_{x \in \mathit{Vars}(l_j)} \sigma(x) \neq \tau_j(x).$$

For the inequations $t_j < s_j$, it is obviously optimal to assign the smallest possible term to every variable. Larger terms only have to be considered due to

the inequalities $l_j \neq r_j$. If there is a grounding σ that satisfies $t_j < s_j$, then any grounding σ' with $\sigma'(x) \leq \sigma(x)$ for all variables x satisfies $t_j < s_j$. Hence, if there exists a solution, then there also exists a solution that only uses the $m+1$ smallest terms for every variable. This is because every inequality $l_j \neq r_j$ only restricts at most one term for every variable, so for every variable the $m+1$ smallest terms contain the smallest term that is not restricted for that variable.

As we only have to consider the $m+1$ smallest terms for every variable, the size of the groundings we have to consider is polynomially bounded by the input size. Let f be the function with the maximal arity and let $p = \text{arity}(f)$. Let a be the smallest constant. We claim that every of the $m+1$ smallest terms has at most $mp+1$ symbols. Proof by contradiction: Assume t_0 is one of the $m+1$ smallest terms with $\#t_0 > mp+1$. Perform the following m times: Obtain t_{i+1} by replacing any subterm $g(s_1, \ldots, s_n)$, where the s_i are constants, by a. The number of symbols decreases by at most p, so $\#t_i > (m-i)p+1$. As none of the t_i is a constant, such a subterm always exists. After m steps, we obtain terms $t_0 > t_1 > \cdots > t_m$ with $\#t_m > (m-m)p+1 = 1$, i.e., t_m is not a constant, so $t_m > a$. This contradicts the fact that t_0 was one of the $m+1$ smallest terms since at least $m+1$ terms are smaller than t_0. Thus, we can guess a grounding and check in polynomial time whether it is a solution. $\square$

Next we propose an algorithm for testing satisfiability of simple right-ground KBO constraints. Of course, by Proposition 6, there already exists an algorithm, but we expect that the following algorithm performs better in practice. Let C be a simple right-ground KBO constraint with n inequations $t_j < s_j$ and m inequalities $l_j \neq r_j$.

Assume that $\mathit{Vars}(\{t_j \mid 1 \leq j \leq n\} \cup \{l_j \mid 1 \leq j \leq m\}) = \{x_1, \ldots, x_k\}$. As explained in the proof of Proposition 6, we only have to consider the $m+1$ smallest terms for the grounding, so to begin, we generate an ordered list S of the $m+1$ smallest terms. This way, a grounding substitution σ corresponds to a vector $\vec{v} \in \mathbb{N}^k$ where $v_i < m+1$ is the index of the term $\sigma(x_i)$ in S, i.e., $S[v_i] = \sigma(x_i)$. Let $\sigma(\vec{v})$ with $\sigma(\vec{v})(x_i) := S[v_i]$ denote the grounding corresponding to the vector $\vec{v}$. Later on, we give a dynamic programming algorithm to compute the k smallest terms for some number k. Actually, we do not directly generate the $m+1$ smallest terms, but start with a constant number of terms and generate more terms as needed.

The algorithm is given by three inference rules that are represented by an abstract rewrite system. They operate on a state which is either $\bot$ or a four-tuple $(T; \vec{v}; F; C)$ where T is a sequence of variables, the *trace*; $\vec{v} \in \mathbb{N}^k$ is a grounding substitution in vector notation, the *current grounding*; F is a set of *forbidden* groundings; and C is a *simple right-ground KBO constraint*. The initial state for a constraint C is $(\varepsilon; (0, \ldots, 0); \emptyset; C)$, i.e., the *trace* is empty, every variable is mapped to the smallest constant and there are no *forbidden* groundings.

We use the following partial ordering $\leq_F$ on groundings: $\vec{v} \leq_F \vec{u}$ iff for all $i \in \{1, \ldots, k\}$ we have $v_i \leq u_i$. By $\mathit{inc}(\vec{v}, i)$ we denote the grounding $\vec{v}'$ with $v'_i = v_i + 1$ and $v'_l = v_l$ for all $l \in \{1, \ldots, k\}$ with $l \neq i$, i.e., the grounding where we increase the term for the variable x_i by one. Analogously, we define $\mathit{dec}(\vec{v}, i)$,

where we instead decrease the term for the variable x_i by one, i.e., $v'_i = v_i - 1$. The two operations *inc* and *dec* are only used when they are well-defined, i.e., they yield a grounding $\vec{v} \in \mathbb{N}^k$ where $v_i < m+1$. The operation *inc* is only used when an inequality $l_j \neq r_j$ is not satisfied, and this can happen at most m times without intermediate Backtrack steps. The operation $dec(\vec{v}, i)$ is only used for Backtrack, and by Lemma 15, in this case $v_i > 0$.

The role of F is that we want to keep the algorithm from considering wrong groundings again. For all $\vec{u} \in F$, we do not visit states with grounding $\vec{v}$ if $\vec{v} \geq_F \vec{u}$. When we Backtrack, we insert the current grounding into F. The trace T records the last updated variables so Backtrack is able to undo the last Increase operation. As will be proven in Theorem 18, the algorithm terminates in $\bot$ iff there exists no solution, and if there exists a solution, then it terminates in a state where the current grounding $\vec{v}$ is a solution.

Increase $\quad (T; \vec{v}; F; C) \quad \Rightarrow_{\text{KCS}} \quad (Tx_i; \vec{v}'; F; C)$

provided $\vec{v}' = inc(\vec{v}, i)$, $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in C$, $l_j\sigma(\vec{v}') \neq r_j$ and there is no $\vec{u} \in F$ with $\vec{v}' \geq_F \vec{u}$

Backtrack $\quad (Tx_i; \vec{v}; F; C) \quad \Rightarrow_{\text{KCS}} \quad (T; \vec{v}'; F \cup \{\vec{v}\}; C)$

provided $\vec{v}' = dec(\vec{v}, i)$ and either

1. $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in C$, but for all $l \in \{1, \ldots, k\}$, we have that $l_j\sigma(inc(\vec{v}, l)) \neq r_j$ implies that there is a $\vec{u} \in F$ with $inc(\vec{v}, l) \geq_F \vec{u}$, or
2. $t_j\sigma(\vec{v}) \geq s_j$ for some $t_j < s_j \in C$

Fail $\quad (\varepsilon; \vec{v}; F; C) \quad \Rightarrow_{\text{KCS}} \quad \bot$

provided either

1. $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in C$, but for all $l \in \{1, \ldots, k\}$, we have that $l_j\sigma(inc(\vec{v}, l)) \neq r_j$ implies that there is a $\vec{u} \in F$ with $inc(\vec{v}, l) \geq_F \vec{u}$, or
2. $t_j\sigma(\vec{v}) \geq s_j$ for some $t_j < s_j \in C$

Informally, Increase is applicable if some inequality $l_j \neq r_j$ is not fulfilled and we can fix this with the new grounding $inc(\vec{v}, i)$ which is not forbidden by F. Backtrack undoes an operation and is applicable if either some inequality $l_j \neq r_j$ is not fulfilled, but Increase is not applicable, or if some inequation $t_j < s_j$ is not fulfilled. Fail is applicable if Backtrack would be applicable on an empty *trace*, i.e., there is no operation to undo.

Obviously, there is no state on which we can apply both Backtrack and Fail.

Definition 7. *A reasonable strategy is a strategy that prefers Backtrack and Fail over Increase.*

Example 8. Consider a signature with constants a, b, c and a binary function f. We set $w(a) = 1; w(b) = w(c) = 2; w(f) = 3$ and $a \prec b \prec c \prec f$. We consider the constraint

$$C = \{x_1 \neq a, f(x_1, x_2) < f(a, c)\}.$$

The $m+1$ smallest terms, where $m = 1$, are a, b. This is the unique execution of the algorithm. In order to increase readability, for $\vec{v}$, we write the terms instead of the indices.

$$\begin{array}{ll}
 & (\varepsilon; (a, a); \emptyset; C) \\
\Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_1; (b, a); \emptyset; C) \\
\Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a, a); \{(b, a)\}; C) \\
\Rightarrow^{\text{Fail}}_{\text{KCS}} & \bot
\end{array}$$

The algorithm terminates in $\bot$, so there is no solution.

Example 9. Consider a signature with constants a, b, a binary function g and a ternary function f. Let $w(a) = 1, w(b) = w(f) = w(g) = 2$ and $a \prec b \prec g \prec f$. The constraint is

$$C = \{x_1 < b, g(x_2, a) < g(b, b), f(x_1, x_2, x_3) \neq f(a, a, a), g(x_1, x_2) \neq g(a, b)\}.$$

The $m + 1$ smallest terms, where $m = 2$, are $a, b, g(a, a)$.

$$\begin{array}{ll}
 & (\varepsilon; (a, a, a); \emptyset; C) \\
\Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_1; (b, a, a); \emptyset; C) \\
\Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a, a, a); \{(b, a, a)\}; C) \\
\Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_2; (a, b, a); \{(b, a, a)\}; C) \\
\Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_2 x_2; (a, g(a, a), a); \{(b, a, a)\}; C) \\
\Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (x_2; (a, b, a); \{(b, a, a), (a, g(a, a), a)\}; C) \\
\Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a, a, a); \{(b, a, a), (a, g(a, a), a), (a, b, a)\}; C) \\
\Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_3; (a, a, b); \{(b, a, a), (a, g(a, a), a), (a, b, a)\}; C)
\end{array}$$

The algorithm has found a solution, so no rule is applicable and it terminates. Note that after the third and fifth operation, we cannot increase x_1 because $(b, b, a) \geq_F (b, a, a) \in F$.

Next we prove the correctness of the algorithm.

Lemma 10. *If* $(\varepsilon; (0, \ldots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}'; F; C)$*, then there is no* $\vec{u} \in F$ *with* $\vec{v}' \geq_F \vec{u}$*.*

Proof. We prove this by induction on l. For $l = 0$, this holds since $F = \emptyset$. For $l > 0$, the last applied rule must have been either Increase or Backtrack. If the last applied rule was Increase, then there cannot be such a $\vec{u}$ because Increase does not modify F and because this is part of the condition of the Increase rule. Now assume the last applied rule was Backtrack, so the previous state was $(Tx_i; \vec{v}; F'; C)$ with $\vec{v}' = dec(\vec{v}, i)$ and $F = F' \cup \{\vec{v}\}$. If there was some $\vec{u}$ in F such that $\vec{v}' \geq_F \vec{u}$, then, since $\vec{v}' <_F \vec{v}$, we have $\vec{v} >_F \vec{u}$. Hence, by the induction hypothesis, $\vec{u} \notin F'$, so as $F = F' \cup \{\vec{v}\}$, it must hold that $\vec{u} = \vec{v}$, contradiction to $\vec{v} >_F \vec{u}$. □

Lemma 11. *If* $(\varepsilon; (0, \ldots, 0); \emptyset; C) \Rightarrow^i_{KCS} (T; \vec{u}; F; C) \Rightarrow^l_{KCS} (T'; \vec{u}'; F'; C)$ *for* $l > 0$*, then* $\vec{u} \neq \vec{u}'$ *or* $F \neq F'$*.*

Proof. If all l rule applications are applications of the Increase rule, then clearly $\vec{u} <_F \vec{u}'$, so in particular, $\vec{u} \neq \vec{u}'$. There is no rule that removes elements from F, so $F \subseteq F'$. If there is at least one application of the Backtrack rule among the l rule applications, the current assignment $\vec{v}$ is added to F, and by Lemma 10, $\vec{v} \notin F$, so F is modified and $F \neq F'$. □

Proposition 12. $\Rightarrow_{KCS}$ *is well-founded, i.e., the algorithm always terminates.*

Proof. By Lemma 11, we can reach every combination of $\vec{v}$ and F at most once. For $\vec{v}$, there are $(m+1)^k$ possibilities. We only add occurring groundings to F, so the number of possibilities for F is upper bounded by the number of subsets of all possible groundings which is $2^{(m+1)^k}$. Thus, the number of reached states is finite (it is at most $(m+1)^k 2^{(m+1)^k}$), so the algorithm terminates. □

Of course, the upper bounds in the proof of Proposition 12 are far too high and the algorithm will run much faster in practice.

Lemma 13. *If* $(\varepsilon; (0, \ldots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}; F; C)$ *and* $\vec{u} \in F$*, then for all* $\vec{u}' \geq_F \vec{u}$ *it holds that* $\vec{u}'$ *cannot be a solution.*

Proof. The proof is by induction on l. For $l = 0$, we have $F = \emptyset$, so this holds. For $l > 0$, if the last applied rule was Increase, the statement follows by the induction hypothesis since F is not modified. Now assume that the last applied rule was Backtrack. Let $(T'; \vec{v}'; F'; C)$ be the previous state. We only have to show that all $\vec{u}' \geq_F \vec{v}'$ cannot be solutions, for all other elements of $F = F' \cup \{\vec{v}'\}$, this follows by the induction hypothesis. First assume that Backtrack is applicable because of condition (1). Then $\vec{v}'$ cannot be a solution since $l_j\sigma(\vec{v}') = r_j$. For $\vec{u}' >_F \vec{v}'$, if $l_j\sigma(\vec{u}') = r_j$, then $\vec{u}'$ clearly cannot be a solution. Otherwise, there is a variable x_i such that $\vec{u}' \geq_F inc(\vec{v}', i)$ and $l_j\sigma(inc(\vec{v}', i)) \neq r_j$. However, it is part of condition (1) that then, there is an element $\vec{u}'' \in F'$ with $\vec{u}'' \leq_F inc(\vec{v}', i) \leq_F \vec{u}'$, so by the induction hypothesis, $\vec{u}'$ cannot be a solution. If Backtrack is applicable because of condition (2), then $t_i\sigma(\vec{v}') \geq s_i$ for some $i \in \{1, \ldots, n\}$. Clearly, if $\vec{u}' \geq_F \vec{v}'$, then also $t_i\sigma(\vec{u}') \geq s_i$, so $\vec{u}'$ cannot be a solution. □

Corollary 14. *If* $(\varepsilon; (0, \dots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}; F; C)$ *and condition (1) or condition (2) of Fail is fulfilled for* $(T; \vec{v}; F)$*, then for all* $\vec{u} \geq_F \vec{v}$*,* $\vec{u}$ *cannot be a solution.*

Proof. The conditions for Fail are the same as the conditions for Backtrack, so this follows by the proof of Lemma 13. □

Lemma 15. *If* $(\varepsilon; (0, \dots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}; F; C)$*, then for all* $i \in \{1, \dots, k\}$ *the number of occurrences of* x_i *on the trace* T *equals* v_i*.*

Proof. In the following, we denote the number of occurrences of x_i in T by $C(T, x_i)$. The proof is by induction on l. If $l = 0$, the statement trivially holds. For $l > 0$ let $(T'; \vec{v}'; F'; C)$ be the previous state. If the last applied rule was Increase, then $T = T'x_i$ and $\vec{v} = inc(\vec{v}', i)$, so

$$C(T, x_i) = C(T', x_i) + 1 \stackrel{\text{IH}}{=} v'_i + 1 = v_i.$$

For $j \neq i$, $C(T, x_j) = C(T', x_j)$ and $v_j = v'_j$, so the statement follows by the induction hypothesis. If the last applied rule was Backtrack, then $T' = Tx_i$ and $\vec{v} = dec(\vec{v}', i)$, so

$$C(T, x_i) = C(T', x_i) - 1 \stackrel{\text{IH}}{=} v'_i - 1 = v_i.$$

Again, for $j \neq i$, $C(T, x_j) = C(T', x_j)$ and $v_j = v'_j$, so the statement follows by the induction hypothesis. □

Lemma 16. *If* $(\varepsilon; (0, \dots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}; F; C) \Rightarrow^{Fail}_{KCS} \perp$*, then there exists no solution.*

Proof. Since Fail is applicable on $(T; \vec{v}; F; C)$, $T = \varepsilon$, so by Lemma 15, $\vec{v} = (0, \dots, 0)$. Hence, by Corollary 14, for all $\vec{u} \geq_F (0, \dots, 0)$, $\vec{u}$ cannot be a solution, so there exists no solution. □

Lemma 17. *If* $(\varepsilon; (0, \dots, 0); \emptyset; C) \Rightarrow^l_{KCS} (T; \vec{v}; F; C)$ *and no rule is applicable on* $(T; \vec{v}; F; C)$ *then* $\vec{v}$ *is a solution.*

Proof. Assume that for some $j \in \{1, \dots, n\}$, we had $t_j\sigma(\vec{v}) \geq s_j$. Then, either Backtrack or Fail would be applicable. Now assume that for some $j \in \{1, \dots, m\}$, we had $l_j\sigma(\vec{v}) = r_j$. Then, either Increase or Backtrack would be applicable. □

Theorem 18. *The algorithm is correct: If there exists a solution, then starting from* $(\varepsilon; (0, \dots, 0); \emptyset; C)$*, the algorithm terminates in a state* $(T; \vec{v}; F; C)$ *where* $\vec{v}$ *is a solution. If there is no solution, the algorithm terminates in* $\perp$*.*

Proof. Follows by Proposition 12, Lemma 16 and Lemma 17. □

We have implemented the above algorithm in the context of the SPASS reasoning workbench. The efficiency of the algorithm depends on the respective variables we choose for Increase. If there exists a solution, then there exists an execution using only the rule Increase. The following criteria might be useful to select the best variable for Increase:

– We prefer variables that do not occur in "critical" inequations, or in a minimal number of inequations. A "critical" inequation is one where the weight difference is 0 or close to 0.
– We prefer variables x_i for which the next term is not restricted by any inequality $l_j \neq r_j$.
– We prefer variables x_i for which the next term does not have a larger weight, or for which the increase in weight is minimal.
– We prefer variables that fix multiple inequalities $l_j \neq r_j$ instead of just one.

It is possible to calculate and maintain some score for every variable here and decide based on this score. The exact selection criteria still need to be further explored.

A remaining problem from the presentation of the algorithm is how to compute the k smallest terms. If the occurring weights are rather small, the following dynamic programming algorithm might be useful in practice. The idea is to compute all terms of a specific weight for increasing weights until we generated at least k terms. Unfortunately, there may be exponentially many terms of a specific weight where the exponent is the maximal arity of a function and the base is the number of terms of smaller weights. However, k is bounded above by the number of inequalities m, the number of terms with smaller weights is bounded above by k and the maximal arity is probably small, so it is to be expected that this is not a big problem.

As it is probably hard to find the next possible weight, we simply always increase the weight by 1 starting by the weight of the smallest constant. Our DP array is two-dimensional, one dimension having the weight and the other dimension having the size of the tuple from 1 to *max_arity*. Actually, it is four-dimensional since every entry is a list of tuples of terms and every tuple is a list of its entries. A tuple of size 1 is just a term of the specific weight. The tuples of larger size are needed for the DP transitions where they serve as argument tuples for the functions. We maintain an array *smallest_terms* that will in the end contain the at least k smallest terms.

We iterate over the weights starting at the weight of the minimal constant. Let *curweight* denote the current weight. The idea is to compute all terms of weight *curweight*, sort them, add them to *smallest_terms*, and proceed with weight $\mathit{curweight} + 1$ if $|\mathit{smallest_terms}|$ is still smaller than k. To do so, if *curweight* is not the smallest weight, we first compute the tuples of size 2 to *max_arity* for the previous weight. This is done via DP: For tuple size i we iterate over the terms $s \in \mathit{smallest_terms}$. Then we iterate over the tuples t of size $i-1$ and weight $\mathit{curweight} - 1 - w(s)$ using the DP array and add (s,t) to the current DP entry. Afterwards, we calculate all terms of weight *curweight* by iterating over all symbols f and all tuples t of size $\text{arity}(f)$ and weight $\mathit{curweight} - w(f)$ using the DP array. Then, the term $f(t)$ has weight *curweight*.

We finish this section by a discussion of potential heuristics, sufficient conditions for a simple right-ground KBO constraint to have a solution. As explained before, every inequality $l_j \neq r_j$ rules out any assignment that satisfies τ_j, the matcher from l_j to r_j. Now assume we have m inequalities and know that there

are more than m solutions for the inequation $t < s$, then one might think that there is a grounding that solves all inequalities $l_j \neq r_j$ and the inequation $t < s$. However, this is not true.

Example 19. Consider a signature with constants a, b and c and a binary function f. The weights are $w(a) = 1; w(b) = w(c) = 2; w(f) = 3$ and we use $a \prec b \prec c \prec f$ as a precedence. Now consider the constraint

$$C = \{x \neq a, f(x,y) < f(a,c)\}.$$

The inequation has two solutions, namely $\{x \mapsto a, y \mapsto a\}$ and $\{x \mapsto a, y \mapsto b\}$. However, it has no solution where x is not mapped to a, so for the overall problem, there is no solution.

So the above sufficient condition needs to refined in order to be correct. However, calculating the number of solutions is again NP-hard.

Proposition 20. *Calculating the number of solutions σ for some right-ground inequation $t < s$ is NP-hard.*

Proof. We reduce from the Unbounded Subset Sum Problem (USSP) which is NP-complete by [7]. Let $s_1, \ldots, s_n, T \in \mathbb{N}^+$. We have to find out whether there are $x_1, \ldots, x_n \in \mathbb{N}$ such that $\sum_{i=1}^{n} x_i s_i = T$, i.e., whether there is a multiset of values from $\{s_1, \ldots, s_n\}$ that sums up to T. Assume we had an oracle that could compute the number of solutions for any inequation $l < r$ where r is ground. We will use this oracle twice.

For both uses, we use a signature with constants c and d and unary functions $f_1, \ldots, f_n$. We have $w(c) = 1$, $w(f_i) = s_i$ for $i \in \{1, \ldots, n\}$ and $d \prec c \prec f_1 \prec \cdots \prec f_n$. For the first case, set $w(d) = T+2$. Using the oracle with the inequation $x < d$, we get the number of terms smaller than d. Since d is the smallest term of weight $T+2$, this is exactly the number of terms with weight $\leq T+1$. For the second case, set $w(d) = T+1$. Again, using the oracle with the inequation $x < d$, we get the number of terms smaller than d. This time, this is the number of terms with weight $\leq T$. If we now subtract those values, we get the number of terms with weight exactly $T+1$.

Now the USSP has a solution iff the number of terms with weight exactly $T+1$ is not 0. Every term t of weight $T+1$ must have the constant c as subterm since the weight of d is too large. The rest of t must consist of the unary functions. Hence, the weights of the unary functions used sum up to $T+1-1 = T$. Since the weights of the unary functions correspond to the numbers from the USSP, this yields a solution for the USSP. Conversely, given a solution to the USSP, we can construct a term of weight $T+1$ analogously. □

The problem with the aforementioned insufficient condition is that an inequality $l_j \neq r_j$ does not necessarily rule out only one grounding, but possibly infinitely many groundings. This happens if there are variables that are not restricted by the matcher τ_j of l_j and r_j. However, the criterion can be refined to a correct sufficient condition. If we restrict ourselves to the $m+1$

smallest terms again, we would again at least have a finite number of groundings that $l_j \neq r_j$ rules out. If we now sum up these numbers over all inequalities, we have an upper bound on the total number of ruled out groundings. For the inequation $t < s$, the same problem with variables that do not occur arises (there may be infinitely many solutions), so here, we restrict ourselves to the $m + 1$ smallest terms again. If now, the number of solutions for $t < s$ is larger than the upper bound on the total number of ruled out groundings, we can actually be sure that there is a solution. However, this correct sufficient condition is hard to compute and therefore seems to be not very useful in practice.

4 Further Constraint Variants and Ordering Relaxation

In this section we study further variants of constraint problems and eventually extend the algorithm of Sect. 3 to alternating KBO constraints.

Proposition 21. *Checking satisfiability for right-ground KBO constraints restricted to strict inequations is NP-hard.*

Proof. The proof strategy is the same as the one used in the proof of Proposition 5. The encoding for positive clauses stays the same as $<$ is still allowed. For negative clauses $\neg P \vee \neg Q \vee \neg R$ we encode them as $f(x_P, x_Q, x_R) > f(a, a, a)$. This inequation can only be satisfied by a grounding that does not map all of these variables to a, and is trivially satisfied by any such grounding. $\square$

In particular, we have seen that only having constraints of the form $t_i < s_i$ and $t_i > s_i$ suffices to make the problem NP-hard. Next we turn to a weaker term ordering $\leq_{\text{sym}}$ solely based on symbol counting. Even for this ordering constraint solving remains NP-hard.

Definition 22. *For ground terms $t, s \in T(\Sigma)$, we define $t \leq_{sym} s :\iff |sym(t)| \leq |sym(s)|$, i.e., t does not contain more symbols than s.*

Definition 23. *A right-ground symbol constraint C is a finite set of atoms $t\#s$ with $t \in T(\Sigma, \mathcal{X})$, $s \in T(\Sigma)$ and $\# \in \{\leq_{sym}, \neq\}$. Satisfiability is defined analogously to the satisfiability of KBO constraints.*

Proposition 24. *Checking satisfiability for right-ground symbol constraints is NP-hard.*

Proof. The proof strategy is the same as the one used in the proof of Proposition 5. We encode positive clauses $P \vee Q \vee R$ as $f(x_P, x_Q, x_R) \leq f(g(a), g(a), a)$. The only way to satisfy this inequation is to map at least one of these variables to a. Negative clauses $\neg P \vee \neg Q \vee \neg R$ are encoded as $f(x_P, x_Q, x_R) \neq f(a, a, a)$. $\square$

In particular, the NP-hardness of these problems is not caused by the complicated structure of the KBO since the problem is already NP-hard for a comparison as simple as counting the number of symbols.

Our next variants are motivated by the definition of congruence classes with respect to terms with variables. For the first variant, all instances of the defining term t have to be smaller than a single ground term β and different from ground terms $s_1, \ldots, s_n$.

Definition 25. *A simple, single ground KBO constraint C consists of terms $t \in T(\Sigma, \mathcal{X})$ and $s_1, \ldots, s_n, \beta \in T(\Sigma)$. We say that C is satisfiable if there exists a substitution σ that is grounding for t such that*

$$\left(\bigwedge_{j=1}^{n} t\sigma \neq s_j \right) \wedge t\sigma < \beta.$$

Proposition 26. *Assuming that we are given the $n+1$ smallest terms, checking satisfiability of simple, single right-ground KBO constraints is in P.*

Proof. Actually, for this problem, if a reasonable strategy (Definition 7) is used, the algorithm from Sect. 3 runs in polynomial time. The key difference to the other problems is that here, every variable occurs in every inequality, so every inequality rules out at most one grounding. We first show that we can only reach polynomially many states. First, consider states $(T; \vec{v}; F; C)$ where $t\sigma(\vec{v}) < \beta$. If $\vec{v}$ does not violate any inequality $t \neq s_i$, then the algorithm terminates, so there is at most one such state. For every inequality $t \neq s_i$, there is at most one grounding $\vec{v}$ that violates it. We claim that we reach at most $k+1$ states with current grounding $\vec{v}$ where k is the number of variables. $\vec{v}$ is reached at most once using Increase because otherwise, there must be an intermediate Backtrack, so $\vec{v}$ would be inserted into F. If we reach $\vec{v}$ using Backtrack for some variable x_i, then $inc(\vec{v}, i)$ was inserted into F, so for every variable x_i, we can reach $\vec{v}$ using Backtrack at most once. Hence, $\vec{v}$ is reached at most $k+1$ times.

Now, consider states $(T; \vec{v}; F; C)$ where $t\sigma(\vec{v}) \geq \beta$. Since a reasonable strategy is used, we must have reached this state from a state that does not violate $t < \beta$, so by the argumentation before, at most k such states can be reached for every inequality, so at most $n \cdot k$ in total where n is the number of inequalities.

Hence, in total, there are at most $(k+1) \cdot n + n \cdot k + 1$ states. The state transitions can be done in polynomial time because we only need to iterate over all inequalities and inequations and over all entries in F. Since there are only polynomially many states and every rule application inserts at most one element into F, F has polynomially many entries. □

Definition 27 (Alternating KBO Constraint). *An alternating KBO constraint C consists of terms $t, s_1, \ldots, s_n \in T(\Sigma, \mathcal{X})$ and $\beta \in T(\Sigma)$. We say that C is satisfiable if there exists a substitution σ that is grounding for t such that for all substitutions τ that are grounding for all s_j we have*

$$\left(\bigwedge_{j=1}^{n} t\sigma \neq s_j\tau \right) \wedge t\sigma < \beta.$$

Proposition 28. *Checking satisfiability for alternating KBO constraints is NP-hard.*

Proof. We reduce from SAT. Let N be a set of clauses and $X_1, \ldots, X_k$ be the variables occurring in N. We use a signature with a k-ary function f, two constants a and b, variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$. Set $t = f(x_1, \ldots, x_k)$. Now for every clause $C_j \in N$ we introduce an inequality $f(x_1, \ldots, x_k) \neq f(s_{j,1}, \ldots, s_{j,k})$ where we set $s_{j,i} = b$ if X_i occurs positively in C_j, $s_{j,i} = a$ if X_i occurs negatively in C_j and $s_{j,i} = y_i$ if X_i does not occur in C_j. The idea is that $x_i = a$ stands for X_i is set to $\top$ and $x_i = b$ stands for X_i is set to $\bot$. $\forall \tau. x_i\sigma \neq y_i\tau$ is obviously impossible to satisfy, so the inequality must be made true by setting some positive variable to a or some negative variable to b.

To ensure that the x_i are only mapped to a or b, we do the following: We first introduce a new constant c and set $\beta = c$. Then we set $w(f) = w(a) = w(b) = 1$, $w(c) = k + 2$ and $c \prec a \prec b \prec f$. If all variables x_i are mapped to a or b, we have $w(t\sigma) = k + 1$, i.e., $t\sigma < \beta$. Any grounding where some x_i is not mapped to a or b results in $t\sigma \geq \beta$.

Now there is a solution σ iff there is a satisfying valuation for N. □

If a reasonable strategy is used, satisfiability of alternating KBO constraints can be checked using the algorithm from Sect. 3. Any solution σ must be such that $t\sigma < \beta$, so we only have to consider instances of the $s_j\tau$ with $s_j\tau < \beta$. What we can now do is to calculate for all s_j all groundings τ with $s_j\tau < \beta$ and add the inequality $t \neq s_j\tau$ to the constraint. There are only finitely many such groundings because we did not allow unary functions f with $w(f) = 0$. This way, we obtain a simple right-ground KBO constraint, so we can apply the algorithm. A more efficient possibility to do this is to add the groundings of the s_j implicitly, i.e., to change the condition of Increase (and the first case of Backtrack and Fail) to whether there exists a matcher τ such that $l_j\sigma(\vec{v}) = r_j\tau$. Also, the condition for the next grounding for Increase changes: It is not that we fix the inequality anymore, but that we change a variable that occurs on the left side of the inequality.

Example 29. Consider the signature $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}\}$, where the superscript numbers denote the function arities, together with the following alternating KBO constraint C:

$$t = f(x_1, x_2) \qquad s_1 = f(g(y_1), y_2)$$
$$\beta = f(f(a, a), a) \qquad s_2 = f(a, a)$$

We set $w(a) = w(g) = w(f) = 1$ and $a \prec g \prec f$. The few smallest terms are

$$a, g(a), g(g(a)), f(a, a).$$

Note that for alternating KBO constraints, it does not suffice anymore to consider the $n + 1$ smallest terms only since an inequality may rule out more than

one term for a variable. However, as mentioned in Sect. 3, we calculate the smallest terms as needed, so this is not a problem. For shorter notation, for F, we omit groundings $\vec{u}$ if there is a grounding $\vec{v} \in F$ with $\vec{v} <_F \vec{u}$. A possible run of the algorithm looks as follows:

$$
\begin{array}{lll}
 & (\varepsilon; (a,a); \emptyset; C) & \\
\Rightarrow_{\text{KCS}}^{\text{Increase}} & (x_1; (g(a),a); \emptyset; C) & s_2,\ \tau = \{\} \\
\Rightarrow_{\text{KCS}}^{\text{Increase}} & (x_1x_1; (g(g(a)),a); \emptyset; C) & s_1,\ \tau = \{y_1 \mapsto a, y_2 \mapsto a\} \\
\Rightarrow_{\text{KCS}}^{\text{Increase}} & (x_1x_1x_1; (f(a,a),a); \emptyset; C) & s_1,\ \tau = \{y_1 \mapsto g(a), y_2 \mapsto a\} \\
\Rightarrow_{\text{KCS}}^{\text{Backtrack}} & (x_1x_1; (g(g(a)),a); \{(f(a,a),a)\}; C) & \beta \\
\Rightarrow_{\text{KCS}}^{\text{Backtrack}} & (x_1; (g(a),a); \{(g(g(a)),a)\}; C) & s_1 \\
\Rightarrow_{\text{KCS}}^{\text{Backtrack}} & (\varepsilon; (a,a); \{(g(a),a)\}; C) & s_1 \\
\Rightarrow_{\text{KCS}}^{\text{Increase}} & (x_2; (a,g(a)); \{(g(a),a)\}; C) & s_2,\ \tau = \{\}
\end{array}
$$

5 Experiments

We implemented the algorithm of Sect. 3 and its extension to constraints with right hand side variables, Definition 27, and tested it in the context of an extended congruence closure (CC) algorithm with variables [6,8,15,16]. We implemented a rather naive variant of [8] with the only goal to generate KBO constraints in order to test our new algorithm on KBO constraints. In contrast to [8] our algorithm considers a finite signature, as usual for first-order logic problems. All experiments were carried out on a Debian Linux server equipped with AMD EPYC 7702 64-Core CPUs running at 3.35GHz and an overall memory of 2TB. The result of all runs as well as all input files and binaries can be found at https://nextcloud.mpi-klsb.mpg.de/index.php/s/BAwd99cxFpSJmSp.

As a first test case we considered all eligible UEQ problems from CASC-J11 [17]. We consider equations and all inequalities for the congruence closure algorithm. The equations generate the congruence and for the inequalities we compute the congruence classes for the respective right and left side term of the inequality. For each example, the KBO function weight was always set to one and the precedence is generated with respect to the occurrence of symbols in the input file in ascending order. For β we chose a fixed nesting depth of 4 and build for each input file a nested term of exactly this depth using function symbols in the order of occurrence in the input, starting with a non-constant function symbol. Out of all eligible problems our CC algorithm terminated on 186 problems within a time limit of 30 min. Please note that although our CC implementation is rather naive, in contrast to the classical ground CC algorithm it does not need a complete grounding; for the examples where our naive algorithm runs out of time a complete grounding is not affordable. The below table shows some typical runs on the UEQ domain. All timings are presented in hundredths of a second and

if they take less than one hundredth of a second we write zero. The below table shows the problem name, the number of ground terms smaller than β indicating the solution space for the constraint, the summed up time of all calls to the KBO constraint solver during the CC run, the number of calls to the KBO constraint solver, and the results of these calls. The three selected examples are typical: most of the problems are satisfiable and the constraint solving algorithm needs almost no time. Note that for the first example all 8014 calls to the constraint solver needed in sum 3 hundreds of a second. The LAT143-1 is the example showing the worst constraint solving performance, i.e., still less than a hundreth of a second per call.

Problem	$< \beta$	Time KBO Constraint Solver	#calls	# true	# false
GRP183-3	9969	3	8014	7946	68
LAT143-1	29720	8797	31033	29554	1479
GRP409-1	103565	0	6	6	0

For the SMT-LIB examples of the UF domain [1], we expanded let operators, removed the typing, coded predicates as equations, did a CNF transformation and then took the first literal of each clause as input for the CC algorithm. Nesting depth was set to 2, the rest done as for the UEQ examples. Removing types means that the number of smaller terms increases, i.e., the problems get potentially more difficult for the constraint solver, in particular for unsatisfiable constraints. The below table again shows some typical results. 1112 examples could be performed by the CC algorithm inside 30 min. The UF domain contains larger examples compared to the UEQ domain, but the characteristics remain. Constraint solving itself takes almost no time. Again all timings are presented in hundredths of a second.

Problem	$< \beta$	Time KBO Constraint Solver	#calls	# true	# false
00336	2120806	0	131	131	0
uf.926761	138397692	0	3882	1988	1894
uf.555113	254939	134	5120	3306	1814

Here uf.555113 is the worst example on constraint solving time with 1.34 s for 5120 calls. Although alternating KBO constraint solving is NP-hard, in practice there are typically only a few inequalities meaning that out of the overall number of terms smaller β, only a few need to be considered.

6 Discussion

We have studied a number of specific KBO constraint solving problems motivated by the SCL calculus and established their complexity. Except for simple,

single right-ground KBO constraints all studied problems are proven NP-hard. We propose an algorithm that eventually runs for alternating KBO constraints which include a quantifier alternation. The algorithm shows nice performance on benchmark problems. Our next step is to turn our naive CC implementation with variables into a robust algorithm.

Acknowledgments. We thank our reviewers for their constructive comments that helped us improve the paper.

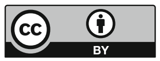
References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). https://www.smt-lib.org/
2. 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
3. Bromberger, M., Gehl, T., Leutgeb, L., Weidenbach, C.: A two-watched literal scheme for first-order logic. 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)
4. 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
5. Bromberger, M., Schwarz, S., Weidenbach, C.: SCL(FOL) revisited (2023). https://doi.org/10.48550/ARXIV.2302.05954. https://arxiv.org/abs/2302.05954
6. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM **27**(4), 758–771 (1980)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Mathematical Sciences Series. Freeman, New York (1979)
8. Hurd, J.: Congruence classes with logic variables. Log. J. IGPL **9**(1), 53–69 (2001)
9. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, I. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
10. Korovin, K., Voronkov, A.: A decision procedure for the existential theory of term algebras with the Knuth-Bendix ordering. In: Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332), pp. 291–302. IEEE (2000)
11. Korovin, K., Voronkov, A.: Knuth-Bendix constraint solving is NP-complete. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 979–992. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_79
12. Korovin, K., Voronkov, A.: Orienting rewrite rules with the Knuth-Bendix order, vol. 183, pp. 165–186. Elsevier (2003)

13. Leidinger, H., Weidenbach, C.: SCL(EQ): SCL for first-order logic with equality. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 228–247. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_14
14. Löchner, B.: Advances in Equational Theorem Proving-Architecture, Algorithms, and Redundancy Avoidance. Dissertation, Fachbereich Informatik, TU Kaiserslautern (2005)
15. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
16. Shostak, R.E.: Deciding combinations of theories. J. ACM **31**(1), 1–12 (1984)
17. Sutcliffe, G.: The CADE ATP system competition - CASC. AI Mag. **37**(2), 99–101 (2016)

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 Critical Pair Criterion for Level-Commutation of Conditional Term Rewriting Systems

Ryota Haga, Yuki Kagaya, and Takahito Aoto(✉)

Niigata University, Niigata, Japan
{r-haga,kagaya}@nue.ie.niigata-u.ac.jp, aoto@ie.niigata-u.ac.jp

Abstract. The rewrite relation of a conditional term rewriting system (CTRS) can be divided into a hierarchy of rewrite relations of term rewriting systems (TRSs) by the depth of the recursive use of rewrite relation in conditions; a CTRS is said to be level-confluent if each of these TRSs are confluent, and level-confluence implies confluence. We introduce level-commutation of CTRSs that extends the notion of level-confluence, in a way similar to extending confluence to commutation, and give a critical pair criterion for level-commutation of oriented CTRSs with extra variables (3-CTRSs). Our result generalizes a criterion for commutation of TRSs of (Toyama, 1987), and properly extends a criterion for level-confluence of orthogonal oriented 3-CTRSs (Suzuki et al., 1995). We also present criteria for level-confluence and commutation of join and semi-equational 3-CTRSs that may have overlaps.

Keywords: Level-commutation · Level-confluence · Commutation · Confluence · Critical pair · Conditional term rewriting systems

1 Introduction

Confluence, which guarantees unique results of computations, is an important property of term rewriting systems (TRSs). Commutativity between two TRSs is a natural generalization of confluence in the sense that self-commutativity coincides with confluence. It also allows to infer confluence of TRSs in a modular way—the union of two confluent TRSs is confluent if they commute.

Conditional term rewriting systems (CTRSs) are extensions of TRSs in which each rewrite rule can be equipped with conditions, where these conditions are supposed to be evaluated recursively using the underlying CTRS itself. Some type of CTRSs is known as a model of functional (and logic) programs. The underlying logic of TRSs is the equational logic, whereas the one of CTRSs is

The parts of this research were done while the first and second authors were students at Niigata University. Partial results of the paper have been appeared in workshops PPL 2020, PPL 2022 and IWC 2022.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 99–116, 2023.
https://doi.org/10.1007/978-3-031-43369-6_6

called the quasi-equational logic, constituting also an important class of systems for reasoning on a wider class of algebras.

From the computational point of view, the rewrite relation of a CTRS can be divided into a hierarchy of rewrite relations of TRSs by the depth of the recursive use of rewrite relation in conditions; a CTRS is said to be level-confluent if each of these TRSs are confluent. Suzuki et al. showed a criterion for orthogonal (i.e. left-linear non-overlapping) oriented CTRSs to be level-confluent [14]. Level-confluence implies confluence, and their result can be thought as a generalization of confluence of orthogonal TRSs. More crucially, since much fewer criterion have been obtained for CTRSs comparing to TRSs, level-confluence can be seen as an important approach to obtain confluence proofs of CTRSs. In contrast to TRSs, where many extensions of the orthogonality criterion for left-linear (possibly overlapping) TRSs to have confluence have been explored (e.g., [4,8, 11,16]), similar extensions for CTRSs are not known. Similarly, several criteria for ensuring commutation for left-linear TRSs are known (e.g., [16,19]). Again, similar criteria for left-linear CTRSs are not known. In this paper, we give a criterion for a class of (possibly overlapping) left-linear oriented CTRSs, under which we prove level-commutation of such CTRSs. Our result is a generalization of the one given for TRSs in [16] and properly extends the result of [14] mentioned above. We also present criteria for level-confluence and commutation of left-linear join and semi-equational CTRSs that may have overlaps.

The rest of the paper is organized as follows. In the next section, we fix some notions and notations used in this paper, and explain two results that give starting points of our work. In Sect. 3, we present our main theorem on level-commutation of *oriented* CTRSs and its proof in detail, and explain relations to the previous results. We then give some results on *join* CTRSs and *semi-equational* CTRSs in Sect. 4. Section 5 concludes.

2 Preliminaries

We basically follow standard notions and notations (e.g., [3,10]). Below, we explain some key notions and fix notations that will be used in this paper, while omitting most of definitions of standard notions and notations.

We consider a set $\mathcal{F}$ of function symbols. The set of variables is denoted by $\mathcal{V}$ and the set of terms over $\mathcal{F}$ and $\mathcal{V}$ is by $\mathrm{T}(\mathcal{F},\mathcal{V})$. We sometimes specify a set $\mathcal{C} \subseteq \mathcal{F}$ of *constructors* to give the set of constructor terms $\mathrm{T}(\mathcal{C},\mathcal{V})$, i.e. terms over $\mathcal{C}$ and $\mathcal{V}$. The set of variables in a term t is denoted by $\mathcal{V}(t)$. A term t is *linear* if each variable occurs in t at most once; t is *ground* if no variable occurs in t. The size of a term t is denoted by $|t|$. The set of positions in a term t is denoted by $\mathrm{Pos}(t)$; the *root* position is written as ϵ. The symbol at a position $p \in \mathrm{Pos}(t)$ in a term t is written as $t(p)$. We put $\mathrm{Pos}_{\mathcal{F}}(t) = \{p \in \mathrm{Pos}(t) \mid t(p) \in \mathcal{F}\}$.

If $t = C[u]_p$ for a context C, we say u is a *subterm* of t (at a position $p \in \mathrm{Pos}(t)$). The subterm of t at a position $p \in \mathrm{Pos}(t)$ is written as $t|_p$. For terms $t = C[u]_p$ and s, the term $C[s]_p$ is denoted by $t[s]_p$. We speak of subterm *occurrences* when we consider subterms with their respective positions; see e.g.

[15] for a precise formalization of subterm occurrences. We will use capital letters $A, B, \ldots$ for subterm occurrences. For simplicity, a subterm occurrence A in a term is also treated as a term A (for example, we might write $\mathcal{V}(A)$). Suppose A, B are subterm occurrences in a term t. If $t = C[A]_p$ and $t = C'[B]_q$ with $p \leq q$ ($p < q$) we say that B is a (proper) subterm occurrence in a subterm occurrence A and write $B \subseteq A$ ($B \subset A$, respectively). Overlaps on subterm occurrences will be used to give a notion of weight on which our induction proof works.

A *term rewriting system* (*TRS*, for short) $\mathcal{R}$ is a set of *rewrite rules*, where each *rewrite rule* $l \to r$ satisfies the conditions $l \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. Rewrite rules are identified modulo renaming. A TRS $\mathcal{R}$ is *left-linear* if l is linear for each $l \to r \in \mathcal{R}$. We write $s \to_{\mathcal{R}}^{p} t$ if $s|_p$ is the redex of this rewrite step; we also write $s \xrightarrow{A}_{\mathcal{R}} t$ to indicate the redex occurrence A of this rewrite step. The relation $\to_{\mathcal{R}}$ over terms is called the *rewrite relation* of $\mathcal{R}$, and its reflexive transitive closure is denoted by $\xrightarrow{*}_{\mathcal{R}}$. A *reduction* is a successive sequence of rewrite steps $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_n$, where n is the *length* of this reduction. When no confusion arises, a reduction $s \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t$ is written as $s \xrightarrow{*}_{\mathcal{R}} t$ for brevity, whose length is denoted by $|s \xrightarrow{*}_{\mathcal{R}} t|$. We have a *parallel rewrite step* $s \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} t$ if $s = C[A_1, \ldots, A_n]$, $t = C[B_1, \ldots, B_n]$ ($n \geq 0$) for some context C and subterm occurrences A_i, B_i such that $A_i \to_{\mathcal{R}}^{\epsilon} B_i$ for all $i = 1, \ldots, n$; this rewrite step is written as $s \xrightarrow{A_1,\ldots,A_n}\!\!\!\!\!\!\!\!\!\!\!\!\!+\quad_{\mathcal{R}} t$ to indicate the redex occurrences $A_1, \ldots, A_n$.

A relation $\to$ is *confluent* if $\xleftarrow{*} \circ \xrightarrow{*} \subseteq \xrightarrow{*} \circ \xleftarrow{*}$; A TRS $\mathcal{R}$ is confluent if so is its rewrite relation $\to_{\mathcal{R}}$. Relations $\to$ and $\leadsto$ *commute* (or, are *commutative*) if $\xleftarrow{*} \circ \overset{*}{\leadsto} \subseteq \overset{*}{\leadsto} \circ \xleftarrow{*}$; TRSs $\mathcal{R}$ and $\mathcal{S}$ commute if so do their rewrite relations $\to_{\mathcal{R}}$ and $\to_{\mathcal{S}}$. Clearly, self-commutativity equals confluence, and from a sufficient criterion for commutativity the one for confluence naturally arises.

Let $l_1 \to r_1$ and $l_2 \to r_2$ be rewrite rules so that their sets of variables are renamed to be disjoint. If a non-variable subterm $l_2|_p$ of l_2 satisfies $l_2|_p\sigma = l_1\sigma$ for some substitution σ, we say that $l_1 \to r_1$ *overlaps on* $l_2 \to r_2$ (at p), provided that $p \neq \epsilon$ for the case $l_1 \to r_1$ and $l_2 \to r_2$ are identical. Suppose $l_1 \to r_1$ overlaps on $l_2 \to r_2$ at p and σ is an mgu of $l_2|_p$ and l_1. Then the pair $\langle l_2[r_1]_p\sigma, r_2\sigma\rangle$ is called a *critical pair* (obtained from that overlap); the pair is called *outer* if $p = \epsilon$ and is called *inner* if $p > \epsilon$. The set of critical pairs from overlaps of rules of $\mathcal{R}$ is denoted by $CP(\mathcal{R})$; the set of outer (inner) critical pairs are denoted by $CP_{out}(\mathcal{R})$ (resp. $CP_{in}(\mathcal{R})$). Let $\mathcal{R}, \mathcal{S}$ be TRSs. The set of critical pairs obtained from overlaps of $l_1 \to r_1 \in \mathcal{R}$ on $l_2 \to r_2 \in \mathcal{S}$ is denoted by $CP(\mathcal{R}, \mathcal{S})$. The sets $CP_{out}(\mathcal{R}, \mathcal{S})$ and $CP_{in}(\mathcal{R}, \mathcal{S})$ are defined similarly. We are now ready to state a sufficient criterion for commutativity of TRSs.

Proposition 1 ([16]). *Let $\mathcal{R}$ and $\mathcal{S}$ be left-linear TRSs. If both of the following conditions are satisfied, then $\mathcal{R}$ and $\mathcal{S}$ commute:*

1. *for any $\langle p, q\rangle \in CP(\mathcal{R}, \mathcal{S})$, $p \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{S}} \circ \xleftarrow{*}_{\mathcal{R}} q$, and*
2. *for any $\langle q, p\rangle \in CP_{in}(\mathcal{S}, \mathcal{R})$, $q \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} p$ holds.*

The above criterion for commutativity arises a criterion for confluence: a left-linear TRS $\mathcal{R}$ is confluent if (1) for any $\langle p, q\rangle \in CP_{out}(\mathcal{R})$, $p \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} \circ \xleftarrow{*}_{\mathcal{R}} q$,

and (2) for any $\langle q, p\rangle \in CP_{in}(\mathcal{R})$, $q \twoheadrightarrow\!\!\!\!\!\!\!\rightarrow_{\mathcal{R}} p$ holds. Note here in the condition (1), considering $\langle p, q\rangle \in CP_{out}(\mathcal{R})$ is sufficient, instead of considering $\langle p, q\rangle \in CP(\mathcal{R})$, because of the presence of condition (2).

A (directed) equation is an ordered pair $\langle u, v\rangle$ of terms, written as e.g. $u \approx v$. A *conditional rewrite rule* has the form $l \to r \Leftarrow u_1 \approx v_1, \ldots, u_k \approx v_k$ where $l \notin \mathcal{V}$; here, $u_1 \approx v_1, \ldots, u_k \approx v_k$ is a sequence of (directed) equations, called the *conditional part* of the rule. Often, we will use a meta-variable, say c, to denote the conditional part of the rule. Let $c = u_1 \approx v_1, \ldots, u_k \approx v_k$. Then, for any given substitution σ, we put $c\sigma = u_1\sigma \approx v_1\sigma, \ldots, u_k\sigma \approx v_k\sigma$. Also, we write e.g. $\mathcal{V}(l, c)$ to denote the set of variables occurring in l and c. We often also treat c as a set $\{u_1 \approx v_1, \ldots, u_k \approx v_k\}$ so as to write $u \approx v \in c$, $c\sigma \subseteq \rightsquigarrow$, etc., whose meaning should be apparent. The empty sequence is also written as $\emptyset$, and $l \to r \Leftarrow \emptyset$ is abbreviated as $l \to r$.

Conditional term rewriting system (CTRS, for short) is a set of conditional rewrite rules. In the literature, CTRSs are categorized into several types of CTRSs according the way of interpreting the conditions of the rules used in the definition of their rewrite steps. A *rewrite step* of *oriented* CTRS $\mathcal{R}$ is defined via the following TRSs $\mathcal{R}_n$ $(n \in \mathbb{N})$, which are inductively given as follows: $\mathcal{R}_0 = \emptyset$, $\mathcal{R}_{n+1} = \{l\sigma \to r\sigma \mid l \to r \Leftarrow c \in \mathcal{R}, c\sigma \subseteq \xrightarrow{*}_{\mathcal{R}_n}\}$. A rewrite step $s \to_{\mathcal{R}} t$ of CTRS $\mathcal{R}$ is given as $s \to_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}_n} t$ for some n. Note that $m \leq n$ implies $\to_{\mathcal{R}_m} \subseteq \to_{\mathcal{R}_n}$. The smallest n such that $s \to_{\mathcal{R}_n} t$ is called the *level* of the rewrite step $s \to_{\mathcal{R}} t$. We also use the notation $\to_{\mathcal{R}_{<n}} = \bigcup_{i<n} \to_{\mathcal{R}_i}$. We will also write $\mathcal{R}_n \vdash c\sigma$ to denote $c\sigma \subseteq \xrightarrow{*}_{\mathcal{R}_n}$. Except Sect. 4, we will only consider oriented CTRSs in this paper, and thus let us postpone to mention about join or semi-equational CTRSs until Sect. 4. A CTRS $\mathcal{R}$ is *level-confluent* if TRSs $\mathcal{R}_n$ are confluent for all $n \geq 0$. One can naturally extend the notion of level-confluence, in the similar way extending confluence to commutation.

Definition 1 (Level-commutation). *CTRSs $\mathcal{R}$ and $\mathcal{S}$ are* level-commutative *if for any $m, n \geq 0$, $\xleftarrow{*}_{\mathcal{R}_m} \circ \xrightarrow{*}_{\mathcal{S}_n} \subseteq \xrightarrow{*}_{\mathcal{S}_n} \circ \xleftarrow{*}_{\mathcal{R}_m}$.*

Clearly, level-commutativity (level-confluence) implies commutativity (resp. confluence), and self-level-commutativity implies level-confluence.

A conditional rewrite rule $l \to r \Leftarrow c$ has *type 1* if $\mathcal{V}(r, c) \subseteq \mathcal{V}(l)$, *type 2* if $\mathcal{V}(r) \subseteq \mathcal{V}(l)$, *type 3* if $\mathcal{V}(r) \subseteq \mathcal{V}(l, c)$, and *type 4* if "true". A CTRS $\mathcal{R}$ has type n if all rules have type n; CTRSs of type n are also referred to as *n-CTRSs*. We will mainly deal with 3-CTRSs below. Variables occurring in r, c which is not contained in $\mathcal{V}(l)$ are often called *extra* variables.

We now explain some notions necessary to give a sufficient criterion for level-confluence [14]. A CTRS $\mathcal{R}$ is *properly oriented* if $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ implies $\mathcal{V}(u_i) \subseteq \mathcal{V}(l) \cup \bigcup_{j=1}^{i-1} \mathcal{V}(v_j)$ for all $1 \leq i \leq k$, for any $l \to r \Leftarrow u_1 \approx v_1, \ldots, u_k \approx v_k \in \mathcal{R}$. A CTRS $\mathcal{R}$ is *right-stable* if, for all $l \to r \Leftarrow u_1 \approx v_1, \ldots, u_k \approx v_k \in \mathcal{R}$, (1) $(\mathcal{V}(l) \cup (\bigcup_{j=1}^{i-1} \mathcal{V}(u_j, v_j)) \cup \mathcal{V}(u_i)) \cap \mathcal{V}(v_i) = \emptyset$ for all $1 \leq i \leq k$ and (2) for any $1 \leq i \leq k$, v_i is either a linear constructor term or a ground $\mathcal{R}_u$-normal form, where the constructors are given by $\mathcal{C} = \mathcal{F} \setminus \{l(\epsilon) \mid l \to r \Leftarrow c \in \mathcal{R}\}$ and the (extended) TRS $\mathcal{R}_u$ is given by $\mathcal{R}_u = \{l \to r \mid l \to r \Leftarrow c \in \mathcal{R}\}$. A CTRS $\mathcal{R}$ is

left-linear if l is linear for all $l \to r \Leftarrow c \in \mathcal{R}$. Let $l_1 \to r_1 \Leftarrow c_1$ and $l_2 \to r_2 \Leftarrow c_2$ be conditional rewrite rules so that their sets of variables are renamed to be disjoint. We say $l_1 \to r_1 \Leftarrow c_1$ overlaps on $l_2 \to r_2 \Leftarrow c_2$ (at p) if a non-variable subterm $l_2|_p$ of l_2 satisfies $l_2|_p\sigma = l_1\sigma$ for some substitution σ, provided that $p \neq \epsilon$ for the case $l_1 \to r_1 \Leftarrow c_1$ and $l_2 \to r_2 \Leftarrow c_2$ are identical. A CTRS $\mathcal{R}$ is *non-overlapping* if there is no overlap between rules of $\mathcal{R}$; A CTRS $\mathcal{R}$ is *orthogonal* if it is left-linear and non-overlapping.

Proposition 2 (**[14]**). *Let $\mathcal{R}$ be an orthogonal, properly oriented, right-stable 3-CTRS. Then, $\overset{*}{\leftarrow}_{\mathcal{R}_m} \circ \overset{*}{\rightarrow}_{\mathcal{R}_n} \subseteq \overset{*}{\rightarrow}_{\mathcal{R}_n} \circ \overset{*}{\leftarrow}_{\mathcal{R}_m}$ for any $m, n \geq 0$. In particular, $\mathcal{R}$ is level-confluent.*

3 Level-Commutation of Oriented CTRSs

Proposition 1 only deals with TRSs but its scope is not limited to orthogonal ones. On the other hand, Proposition 2 can deal with CTRSs (not only TRSs) but limited to only orthogonal case. Also Proposition 2 only claims on (level-)confluence, whereas Proposition 1 claims on commutativity. A natural question is whether we can unify these two propositions and how—we will focus on this question in the this section.

Our basic idea is to unify proofs of [16, Theorem 3.1] and [14, Theorem 4.6]. The basic scenario of the former proof is showing that $\mathrel{\leftarrow\!\!\shortparallel\!\!-}_{\mathcal{R}} \circ \mathrel{-\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}} \subseteq \mathrel{-\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}} \circ \overset{*}{\mathrel{\leftarrow\!\!\shortparallel\!\!-}}_{\mathcal{R}}$. In the latter, an extended parallel rewriting $\mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{R}_n}$ of $\mathrel{-\!\!\shortparallel\!\!\rightarrow}_{\mathcal{R}}$ was introduced and they showed $\mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{R}_n} \subseteq \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{R}_n} \circ \mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}_{\mathcal{R}_m}$. Naturally, our first attempt was to prove $\mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}}_{\mathcal{R}_m}$. Examining the details, however, it turned out that this scenario does not work (induction does not work). Thus, our first key ingredient is to modify our proof scenario as showing:

$$\mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\shortparallel\!\!\rightarrow}}_{\mathcal{S}_{<n}} \circ \overset{*}{\mathrel{\leftarrow\!\!\shortparallel\!\!\hookleftarrow}}_{\mathcal{R}_m} \qquad (*)$$

We now reason why this scenario is sound using an abstract setting.

Let $(\to_n)_{n \in \mathbb{N}}$ be an $\mathbb{N}$-indexed family of relations on a set X. We put $\to_{<n} = \bigcup_{i<n} \to_i$. We say $(\to_n)_{n \in \mathbb{N}}$ is *up-simulated* if $\overset{*}{\to}_{<n} \subseteq \to_n$ for any $n \in \mathbb{N}$.

Lemma 1. *Let $(\to_n)_{n \in \mathbb{N}}, (\rightsquigarrow_n)_{n \in \mathbb{N}}$ be up-simulated families of relations on a set X. Suppose that[1], for any $m, n \in \mathbb{N}$, $\leftarrow_m \circ \rightsquigarrow_n \subseteq \rightsquigarrow_n \circ \overset{*}{\rightsquigarrow}_{<n} \circ \overset{*}{\leftarrow}_m$. Then, for any $m, n \in \mathbb{N}$, we have (1) $\overset{*}{\leftarrow}_m \circ \rightsquigarrow_n \subseteq \rightsquigarrow_n \circ \overset{*}{\rightsquigarrow}_{<n} \circ \overset{*}{\leftarrow}_m$, (2) $\overset{*}{\leftarrow}_m \circ \rightsquigarrow_n \subseteq \overset{*}{\rightsquigarrow}_n \circ \overset{*}{\leftarrow}_m$ and (3) $\overset{*}{\leftarrow}_m \circ \overset{*}{\rightsquigarrow}_n \subseteq \overset{*}{\rightsquigarrow}_n \circ \overset{*}{\leftarrow}_m$.*

Proof. Use induction. Use (1) to show (2), and then (2) to (3). $\square$

[1] The criterion has some similarity with the *decreasing diagrams*; however, because multiple $\to_m$-steps are allowed, it is not at all apparent (currently, to the authors) whether the criterion can be obtained via the decreasing diagrams.

Now let us adopt our abstract framework to CTRSs. Let $\mathcal{R}$ be a CTRS. The notion of extended parallel rewriting [14] is given as follows: we write $s \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{R}_n} t$ if $s = C[A_1, \ldots, A_p]$, $t = C[B_1, \ldots, B_p]$ $(p \geq 0)$ for some context C and subterm occurrences A_i, B_i such that either $A_i \to^{\epsilon}_{\mathcal{R}_n} B_i$ or $A_i \xrightarrow{*}_{\mathcal{R}_{<n}} B_i$ for all $i = 1, \ldots, p$. We put $\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{R}} = \bigcup_{n \geq 0} \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{R}_n}$, which is called the *extended parallel rewrite step* of $\mathcal{R}$. We will also write $s \overset{A_1,\ldots,A_p}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{R}} t$ to indicate subterm occurrences $A_1, \ldots, A_p$.

Then, from the Lemma 1, it easily follows:

Lemma 2. *Let $\mathcal{R}, \mathcal{S}$ be CTRSs. Suppose $\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_{<n}} \circ \xleftarrow{*}_{\mathcal{R}_m}$ for any $m, n \geq 0$. Then, for any m, n, we have $\overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} \subseteq \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m}$. Hence, for any m, n, we have $\xleftarrow{*}_{\mathcal{R}_m} \circ \xrightarrow{*}_{\mathcal{S}_n} \subseteq \xrightarrow{*}_{\mathcal{S}_n} \circ \xleftarrow{*}_{\mathcal{R}_m}$.*

Proof. Suppose $t_1 \xleftarrow{*}_{\mathcal{R}_m} t \xrightarrow{*}_{\mathcal{S}_n} t_2$. As $\to_{\mathcal{R}_k} \subseteq \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{R}_k}$ for each k we have $t_1 \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} t \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{R}_n} t_2$ (and similarly for $\mathcal{S}$). From the fact $\to_{\mathcal{R}_m} \subseteq \to_{\mathcal{R}_n}$ for $m < n$, it immediately follows that $(\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_n)_{n \in \mathbb{N}}$ is up-simulated (again, similarly for $\mathcal{S}$). Thus, it follows $t_1 \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} t' \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} t_2$ by using Lemma 1 and our hypothesis. Because $\overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{R}_k} \subseteq \xrightarrow{*}_{\mathcal{R}_k}$ for each k (and similarly for $\mathcal{S}$), we obtain $t_1 \xrightarrow{*}_{\mathcal{S}_n} t' \xleftarrow{*}_{\mathcal{R}_m} t_2$. □

It is now concluded from this lemma that our proof scenario $(*)$ works to obtain the level-confluence.

For our proof below, we need to use the induction hypothesis to claim a more general statement as in the above. The following lemma is presented for this purpose.

Lemma 3. *Let $\mathcal{R}, \mathcal{S}$ be CTRSs and $k \in \mathbb{N}$. Suppose $\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_{<n}} \circ \xleftarrow{*}_{\mathcal{R}_m}$ for any m, n such that $m + n < k$. Then, for any m, n such that $m + n < k$, we have (1) $\overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_{<n}} \circ \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m}$, (2) $\overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} \circ \mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}_{\mathcal{S}_n} \subseteq \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m}$ and (3) $\overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} \subseteq \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\shortparallel\;}}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\;\shortparallel\!\!\!\!\!\hookleftarrow}}_{\mathcal{R}_m}$.*

Proof. Use an abstract version of the lemma, which can be proved in the way similar to Lemma 1. □

Our second key ingredient is the following alternative definition of conditional critical pairs.

Definition 2 (Condition-separated CCP). *Suppose $l_1 \to r_1 \Leftarrow c_1$ overlaps on $l_2 \to r_2 \Leftarrow c_2$ at p and σ is an mgu of $l_2|_p$ and l_1. Then the quadruple $\langle l_2[r_1]_p\sigma, r_2\sigma \rangle \Leftarrow \langle c_1\sigma, c_2\sigma \rangle$ is called a* (condition-separated) conditional critical pair (CCP, for short) *(obtained from that overlap); when $p = \epsilon$, the pair is called* outer *and $p > \epsilon$, the pair is called* inner*. The set of (outer, inner) critical pairs obtained from overlaps of $l_1 \to r_1 \Leftarrow c_1 \in \mathcal{R}$ on $l_2 \to r_2 \Leftarrow c_2 \in \mathcal{S}$ is denoted by $CCP(\mathcal{R}, \mathcal{S})$ (resp. $CCP_{out}(\mathcal{R}, \mathcal{S})$, $CCP_{in}(\mathcal{R}, \mathcal{S})$). The set of (outer, inner) critical pairs from overlaps of rules of $\mathcal{R}$ is denoted by $CCP(\mathcal{R})$ (resp. $CCP_{out}(\mathcal{R})$, $CCP_{in}(\mathcal{R})$).*

In most literature, we see that instead of distinguishing two sequences $c_1\sigma$ and $c_2\sigma$, the combined sequence of $c_1\sigma$ and $c_2\sigma$ is employed in the definition of CCPs. But, in our case where CTRSs $\mathcal{R}$ and $\mathcal{S}$ may be different, this distinction is important to state a precise condition of our theorem.

We now present one more preparation: the following lemma is used several times as a part of the proof of our main theorem—when the lemma is used in the proof of our main theorem, the assumption (†) of the lemma can be inferred from the induction hypothesis (of the proof of the main theorem), using Lemma 3.

Lemma 4. *Let $\mathcal{R}$ and $\mathcal{S}$ be 3-CTRSs and suppose that $\mathcal{R}$ is left-linear and right-stable. Suppose that $M = l\sigma$, $N = r\sigma$, $\mathcal{R}_{m-1} \vdash c\sigma$ with $l \to r \Leftarrow c \in \mathcal{R}$. Assume moreover that $M \overset{P_1,\ldots,P_p}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_n} P$ and $P_1, \ldots, P_p$ occurs in the substitution σ. Assume that (†) $\overset{*}{\mathrel{\hookleftarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{R}_i} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_j} \subseteq \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_j} \circ \overset{*}{\mathrel{\hookleftarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{R}_i}$ for any i, j such that $i + j < m + n$. Then, there exists Q such that $N \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_{\mathcal{S}_n} Q$ and $P \to_{\mathcal{R}_m} Q$.*

Now we present our critical pair criterion for commutativity.

Theorem 1. *Let $\mathcal{R}$ and $\mathcal{S}$ be left-linear, properly oriented, right-stable 3-CTRSs. If the following conditions are satisfied, then $\mathcal{R}$ and $\mathcal{S}$ are level-commutative:*

1. *for any $\langle u, v\rangle \Leftarrow \langle c, c'\rangle \in CCP(\mathcal{R}, \mathcal{S})$, $m, n \geq 1$ and substitution ρ, if $c\rho \subseteq \overset{*}{\to}_{\mathcal{R}_{m-1}}$ and $c'\rho \subseteq \overset{*}{\to}_{\mathcal{S}_{n-1}}$ then $u\rho \mathrel{-\!\!\Vert\!\!\!\rightarrow}_{\mathcal{S}_n} \circ \overset{*}{\to}_{\mathcal{S}_{<n}} \circ \overset{*}{\leftarrow}_{\mathcal{R}_m} v\rho$, and*
2. *for any $\langle v, u\rangle \Leftarrow \langle c', c\rangle \in CCP_{in}(\mathcal{S}, \mathcal{R})$, $m, n \geq 1$ and substitution ρ, if $c\rho \subseteq \overset{*}{\to}_{\mathcal{R}_{m-1}}$ and $c'\rho \subseteq \overset{*}{\to}_{\mathcal{S}_{n-1}}$ then $v\rho \mathrel{-\!\!\Vert\!\!\!\rightarrow}_{\mathcal{R}_m} \circ \overset{*}{\to}_{\mathcal{R}_{<m}} u\rho$.*

Proof. Let $M \overset{A_1,\ldots,A_{\bar{m}}}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{R}_m} N$ and $M \overset{B_1,\ldots,B_{\bar{n}}}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_n} P$. We show $N \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_{<n}} Q$ and $P \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{R}_m} Q$ for some Q. For the rewrite steps used in the critical pairs conditions above, note that $\mathrel{-\!\!\Vert\!\!\!\rightarrow}_\ell \circ \overset{*}{\to}_{<\ell} = \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_\ell \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{<\ell}$ as well as $\overset{*}{\to}_\ell = \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_\ell$ for any ℓ. Let Γ and Δ be sets of subterm occurrences in the term M given as follows:

$$\Gamma = \{A_i \mid \exists B_j.\ A_i \subset B_j\} \cup \{B_i \mid \exists A_j.\ B_i \subseteq A_j\}$$
$$\Delta = \{A_i \mid \forall B_j.\ A_i \not\subset B_j\} \cup \{B_i \mid \forall A_j.\ B_i \not\subseteq A_j\}$$

Thus, Γ consists of subterm occurrences A_i's that is a proper subterm occurrence of some B_j and subterm occurrences B_j's that is a subterm occurrence of some A_i; Δ consists of subterm occurrences A_i's and B_j's not contained in Γ. Clearly, for any $1 \leq i \leq \bar{m}$, either one of $A_i \in \Gamma$ or $A_i \in \Delta$ holds, and for any $1 \leq j \leq \bar{n}$, either one of $B_j \in \Gamma$ or $B_j \in \Delta$ holds. In the case $A_{i'}$ and $B_{j'}$ are the same subterm occurrence, we put $A_{i'}$ to Δ and $B_{j'}$ to Γ.

Δ denotes the set of maximal redexes occurrences in the following sense. Let $\Delta = \{M_1, \ldots, M_{\bar{p}}\}$. Then we have $M = C[M_1, \ldots, M_{\bar{p}}]$ for some context C. Furthermore, we have $N = C[N_1, \ldots, N_{\bar{p}}]$ and $P = C[P_1, \ldots, P_{\bar{p}}]$ for some $N_1, \ldots, N_{\bar{p}}, P_1, \ldots, P_{\bar{p}}$ such that $M_i \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_{\mathcal{R}_m} N_i$, $M_i \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_{\mathcal{S}_n} P_i$ $(i = 1, \ldots, \bar{p})$. Thus, it suffices to show for each M_i, there exists Q_i such that $N_i \mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{S}_{<n}} Q_i$ and $P_i \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\!\!\Vert\;\;}}_{\mathcal{R}_m} Q_i$. On the other hand, Γ is used to count the size of overlaps and

is used to give the induction weight. Let $|\Gamma| = \sum_{D\in\Gamma} |D|$. Our proof proceeds on induction on lexicographic combination of $\langle m+n, |\Gamma|\rangle$.

The cases for $m = 0$ or $n = 0$ are easy, thus we consider the cases for $m > 0, n > 0$. We distinguish two cases:

1. Case $M_i \notin \{B_1, \ldots, B_{\bar{n}}\}$. Note that $M_i \in \{A_1, \ldots, A_{\bar{m}}\}$ and $M_i \subseteq B_j$ for no B_j. Let $\{B'_1, \ldots, B'_{\bar{q}}\} = \{B_j \mid 1 \le j \le \bar{n}, B_j \subset M_i\}$. Then we have $M_i = C_i[B'_1, \ldots, B'_{\bar{q}}]$ and $P_i = C_i[\tilde{B}'_1, \ldots, \tilde{B}'_{\bar{q}}]$ so that $M_i \overset{M_i}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_m} N_i$ and $M_i \overset{B'_1,\ldots,B'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$. We distinguish the cases.
   (a) Case $M_i \xrightarrow{*}_{\mathcal{R}_{m-1}} N_i$. Since $\xrightarrow{*}_{\mathcal{R}_{m-1}} \subseteq \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_{m-1}}$, we have $M_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} N_i$. Thus, the desired Q_i is obtained by induction hypothesis and Lemma 3.
   (b) Case $M_i \xrightarrow{M_i}_{\mathcal{R}_m} N_i$. Then $M_i = l\theta$, $N_i = r\theta$ and $\mathcal{R}_{m-1} \vdash c\theta$ for some $l \to r \Leftarrow c \in \mathcal{R}$ and θ. If all redex occurrences B'_j in M_i are contained in the substitution θ, then the desired Q_i exists by Lemmas 3, 4 and induction hypothesis. Suppose otherwise, i.e. there exists B'_j which is not contained in θ. Let $X = \{B'_j \mid 1 \le j \le \bar{q},\ B'_j \text{ is not contained in } \theta\ \}$ and $Y = \{B'_j \mid 1 \le j \le \bar{q},\ B'_j \text{ is contained in } \theta\ \}$. For each $B'_j \in X$, either $B'_j \xrightarrow{B'_j}_{\mathcal{S}_n} \tilde{B}'_j$ or $B'_j \xrightarrow{*}_{\mathcal{S}_{<n}} \tilde{B}'_j$. We distinguish two cases.
      i Case that there exists $B'_j \in X$ such that $B'_j \xrightarrow{B'_j}_{\mathcal{S}_n} \tilde{B}'_j$. W.l.o.g. suppose $j = 1$, i.e. $B'_1 \in X$ and $B'_1 \xrightarrow{B'_1}_{\mathcal{S}_n} \tilde{B}'_1$. Let $M_i \xrightarrow{B'_1}_{\mathcal{S}_n} \tilde{M}_i$. Note also here $\tilde{M}_i \overset{B'_2,\ldots,B'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$. The proof of this case is illustrated in Fig. 1. Let $l' \to r' \Leftarrow c' \in \mathcal{S}$, $B'_1 = l'\theta'$ and $\mathcal{S}_{n-1} \vdash c'\theta'$. Then, since B'_1 is not contained in θ, $l \to r \Leftarrow c \in \mathcal{R}$ and $l' \to r' \Leftarrow c' \in \mathcal{S}$ overlap. Furthermore, as $B'_1 \subset M_i$, we have $\langle v, u\rangle \Leftarrow \langle c', c\rangle \in CCP_{in}(\mathcal{S}, \mathcal{R})$ and there exists a substitution θ'' such that $\tilde{M}_i = v\theta''$ and $N_i = u\theta''$. By our critical pair condition (2), we obtain $\tilde{M}_i \hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow_{\mathcal{R}_m} \tilde{Q}_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_{<m}} N_i$; let $\tilde{M}_i \overset{C_1,\ldots,C_{\bar{r}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_m} \tilde{Q}_i$. Let $\Gamma' = \{C_i \mid \exists B'_j (j \neq 1).\ C_i \subset B'_j\} \cup \{B'_i \mid i \neq 1, \exists C_j.\ B'_i \subseteq C_j\}$. Occurrences in Γ' are distinct, and for any $\tilde{B} \in \Gamma'$, there exists B'_j $(2 \le j \le \bar{q})$ such that $\tilde{B} \subseteq B'_j$. Thus, $|\Gamma'| \le \sum_{j=2}^{\bar{q}} |B'_j|$ holds. Hence, we obtain $|\Gamma'| \le \sum_{j=2}^{\bar{q}} |B'_j| < \sum_{j=1}^{\bar{q}} |B'_j| \le |\Gamma|$. Thus, one can apply induction hypothesis to $\tilde{Q}_i \overset{C_1,\ldots,C_{\bar{r}}}{\leftarrow\!\!\!+\!\!\!\!\!\hookleftarrow}_{\mathcal{R}_m} \tilde{M}_i \overset{B'_2,\ldots,B'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$ so as to obtain $\tilde{Q}'_i, \tilde{P}_i$ such that $\tilde{Q}_i \hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow_{\mathcal{S}_n} \tilde{Q}'_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_{<n}} \tilde{P}_i$ and $P_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_m} \tilde{P}_i$. Since we have $N_i \overset{*}{\leftarrow\!\!\!+\!\!\!\!\!\hookleftarrow}_{\mathcal{R}_{<m}} \tilde{Q}_i \hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow_{\mathcal{S}_n} \tilde{Q}'_i$, by applying induction hypothesis and Lemma 3, it follows that there exists $\tilde{N}_i$ such that $N_i \hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow_{\mathcal{S}_n} \circ \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_{<n}} \tilde{N}_i$ and $\tilde{Q}'_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_{<m}} \tilde{N}_i$. Then, by induction hypothesis and Lemma 3, it follows that there exists Q_i such that $\tilde{N}_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{<\mathcal{S}_n} Q_i$ and $\tilde{P}_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{R}_{<m}} Q_i$.
      ii Case that $B'_j \xrightarrow{*}_{\mathcal{S}_{n-1}} \tilde{B}'_j$ holds for any $B'_j \in X$. As $M_i \overset{B'_1,\ldots,B'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$ and $B'_1, \ldots, B'_{\bar{q}}$ are parallel, we can first rewrite all $B'_j \in Y$ $(1 \le$

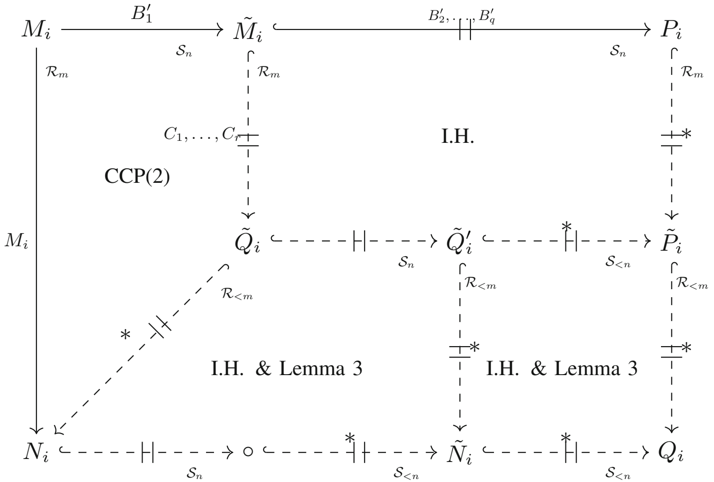


Fig. 1. Case 1.(b).i

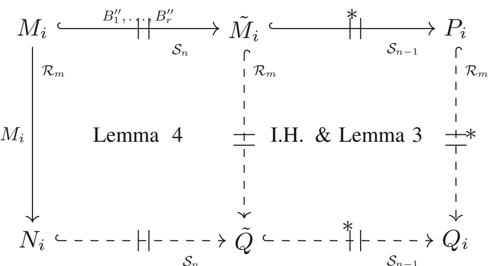


Fig. 2. Case 1.(b).ii

$j \leq \bar{q}$). Namely, let $Y = \{B''_1, \ldots, B''_{\bar{r}}\}$, and we have $M_i \overset{B''_1,\ldots,B''_{\bar{r}}}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_n} \tilde{M}_i \overset{*}{\rightarrow}_{\mathcal{S}_{n-1}} P_i$. The proof of this case is illustrated in Fig. 2. Here, since each B''_j in contained in the substitution θ, one can use Lemma 4 to obtain $\tilde{Q}$ such that $N_i \hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow_{\mathcal{S}_n} \tilde{Q}$ and $\tilde{M}_i \rightarrow_{\mathcal{R}_m} \tilde{Q}$. Now, since $\rightarrow_{\mathcal{R}_m} \subseteq \hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow_{\mathcal{R}_m}$ and $\overset{*}{\rightarrow}_{\mathcal{S}_{n-1}} \subseteq \overset{*}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}}$, we have $\tilde{Q} \leftarrow\!\!\!\!\!+\!\!\!\!\hookleftarrow_{\mathcal{R}_m} \tilde{M}_i \overset{*}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} P_i$. Then, using induction hypothesis and Lemma 3, we can obtain Q_i such that $\tilde{Q} \overset{*}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} Q_i$, $P_i \overset{*}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_m} Q_i$. As a side remark, we mention that our first key ingredient becomes necessary to solve this case.

2. Case $M_i \in \{B_1, \ldots, B_{\bar{n}}\}$. Let $\{A'_1, \ldots, A'_{\bar{q}}\} = \{A_j \mid 1 \leq j \leq \bar{n}, A'_j \subseteq M_i\}$. Then one can put $M_i = C_i[A'_1, \ldots, A'_{\bar{q}}]$, $N_i = C_i[\tilde{A}'_1, \ldots, \tilde{A}'_{\bar{q}}]$, $M_i \overset{A'_1,\ldots,A'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_m} N_i$ and $M_i \overset{M_i}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$. By definition, $M_i \overset{M_i}{\hookrightarrow\!\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_n} P_i$ is either of the form $M_i \overset{*}{\rightarrow}_{\mathcal{S}_{n-1}} P_i$ or $M_i \overset{M_i}{\rightarrow}_{\mathcal{S}_n} P_i$.

Suppose $M_i \xrightarrow{*}_{\mathcal{S}_{n-1}} P_i$. Then, we have $M_i \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} P_i$ and thus the desired Q_i exists by induction hypothesis and Lemma 3.

Thus, it remains to consider the case $M_i \xrightarrow{M_i}_{\mathcal{S}_n} P_i$. Then there exists $l' \to r' \Leftarrow c' \in \mathcal{S}$ and θ' such that $M_i = l'\theta'$, $P_i = r'\theta'$ and $c'\theta' \subseteq \xrightarrow{*}_{\mathcal{S}_{n-1}}$. We distinguish whether all redex occurrences A'_j in M_i are contained in θ' or not. If all redex occurrences A'_j in M_i are contained in θ', then using $\to_{\mathcal{S}_n} \subseteq \hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow_{\mathcal{S}_n} \circ \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{S}_{<n}}$ and $\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow_{\mathcal{R}_m} \subseteq \xrightarrow{*}_{\mathcal{R}_m}$, one obtains desired Q_i by Lemma 4.

So, let us consider there exists A'_j which is not contained in θ'. Let $X' = \{A'_j \mid 1 \leq j \leq \bar{q}, A'_j \text{ is not contained in } \theta'\}$ and $Y' = \{A'_j \mid 1 \leq j \leq \bar{q}, A'_j \text{ is contained in } \theta'\}$. Then for each $A'_j \in X'$, we have either $A'_j \xrightarrow{A'_j}_{\mathcal{R}_m} \tilde{A}'_j$, or $A'_j \xrightarrow{*}_{\mathcal{R}_{<m}} \tilde{A}'_j$. We distinguish two cases.

(a) Case that $A'_j \xrightarrow{A'_j}_{\mathcal{R}_m} \tilde{A}'_j$ for some $A'_j \in X'$. W.l.o.g. assume $j = 1$, i.e. $A'_1 \in X'$ and $A'_1 \xrightarrow{A'_1}_{\mathcal{R}_m} \tilde{A}'_1$. Then there exists $l \to r \Leftarrow c \in \mathcal{R}$ such that $A'_1 = l\theta$ and $c\theta \subseteq \xrightarrow{*}_{\mathcal{R}_{m-1}}$. We further distinguish two cases: (α) the case $A'_1 = M_i$ and $l \to r \Leftarrow c \in \mathcal{R}$ are $l' \to r' \Leftarrow c' \in \mathcal{S}$ are identical, and (β) the case $A'_1 \neq M_i$ or $l \to r \Leftarrow c \in \mathcal{R}$ and $l' \to r' \Leftarrow c' \in \mathcal{S}$ are distinct. We remark that a construction similar to the one in [14] will be used in case of (α) and that our assumption that $\mathcal{R}$ and $\mathcal{S}$ are properly oriented and right-stable will be used here.

   i Case (α). Then we have $M_i = A'_1 \xrightarrow{A'_1}_{\mathcal{R}_m} \tilde{A}'_1 = N_i$ and $M_i \xrightarrow{M_i}_{\mathcal{S}_n} P_i$. By $l\theta = M_i = l\theta'$, $x\theta = x\theta'$ for any $x \in \mathcal{V}(l)$. We also have $\mathcal{R}_{m-1} \vdash c\theta$ and $\mathcal{S}_{n-1} \vdash c\theta'$. Thus, if $\mathcal{V}(r) \subseteq \mathcal{V}(l)$, then $r\theta = r\theta'$, and it suffices to take $r\theta$ as Q_i. Suppose otherwise, i.e. $\mathcal{V}(r) \nsubseteq \mathcal{V}(l)$. Below, let $c = s_1 \approx t_1, \ldots, s_j \approx t_j$ and $c_k = s_1 \approx t_1, \ldots, s_k \approx t_k$ $(1 \leq k \leq j)$. We now show there are substitution ρ_k $(k \in \{0, \ldots, j\})$ satisfying the following properties (a)–(c) by induction.
   - **(a)** $\rho_k = \theta = \theta'$ $[\mathcal{V}(l)]$.
   - **(b)** $dom(\rho_k) \subseteq \mathcal{V}(l) \cup \mathcal{V}(c_k)$.
   - **(c)** for any $x \in \mathcal{V}(l) \cup \mathcal{V}(c_k)$, we have $x\theta' \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} x\rho_k$ and $x\theta \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} x\rho_k$.

   If $k = 0$ then take $\rho_0 = \theta|_{\mathcal{V}(l)}$, and (a)–(c) follow. Suppose $k > 0$. Since r contains an extra variable and $\mathcal{R}$ (or $\mathcal{S}$) is properly oriented, we have $\mathcal{V}(s_k) \subseteq \mathcal{V}(l) \cup \mathcal{V}(c_{k-1})$. Thus, by induction hypothesis on (c), we have $s_k\theta \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} s_k\rho_{k-1}$ and $s_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} s_k\rho_{k-1}$. Furthermore, we have $s_k\theta \xrightarrow{*}_{\mathcal{R}_{m-1}} t_k\theta$ and $s_k\theta' \xrightarrow{*}_{\mathcal{S}_{n-1}} t_k\theta'$ by $\mathcal{R}_{m-1} \vdash c\theta$ and $\mathcal{S}_{n-1} \vdash c\theta'$, respectively. Hence, $s_k\rho_{k-1} \overset{*}{\leftarrow\!\!\!\!\parallel\!\!\!\!\!\hookleftarrow}_{\mathcal{S}_{n-1}} s_k\theta \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} t_k\theta$ and $t_k\theta' \overset{*}{\leftarrow\!\!\!\!\parallel\!\!\!\!\!\hookleftarrow}_{\mathcal{S}_{n-1}} s_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} s_k\rho_{k-1}$. Then, by applying induction hypothesis and Lemma 3, we obtain q', r' such that $s_k\rho_{k-1} \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} q' \overset{*}{\leftarrow\!\!\!\!\parallel\!\!\!\!\!\hookleftarrow}_{\mathcal{S}_{n-1}} t_k\theta$ and $t_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} r' \overset{*}{\leftarrow\!\!\!\!\parallel\!\!\!\!\!\hookleftarrow}_{\mathcal{S}_{n-1}} s_k\rho_{k-1}$. Thus, one obtains $r' \overset{*}{\leftarrow\!\!\!\!\parallel\!\!\!\!\!\hookleftarrow}_{\mathcal{S}_{n-1}} s_k\rho_{k-1} \overset{*}{\hookrightarrow\!\!\!\!\!\parallel\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} q'$. Again, by applying induction hypoth-

esis and Lemma 3, we obtain s' such that $r' \xhookrightarrow{*}\!\!\!\!\!\!\!+\!\!\!\!\!\rightarrow_{\mathcal{R}_{m-1}} s' \xleftarrow{*}\!\!\!\!+\!\!\!\hookrightarrow_{\mathcal{S}_{n-1}} q'$. Thus, we have $t_k\theta \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} s'$ and $t_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} s'$.

We know that t_k is either a ground $\mathcal{R}_u$-normal form or a linear constructor term (w.r.t. $\mathcal{R}$) by the right-stability of $\mathcal{R}$, and that t_k is either a ground $\mathcal{S}_u$-normal form or a linear constructor term (w.r.t. $\mathcal{S}$) by the right-stability of $\mathcal{R}$. Suppose t_k is a ground $\mathcal{R}_u$-normal form or t_k is a ground $\mathcal{S}_u$-normal form. Then, $t_k\theta' = t_k\theta = t_k$ by $\mathcal{V}(t_k) = \emptyset$, and thus, $t_k = s'$ by $t_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} s'$. Furthermore, as we are assuming $\mathcal{V}(r) \nsubseteq \mathcal{V}(l)$, we know $\mathcal{V}(s_i) \subseteq \mathcal{V}(l) \cup \mathcal{V}(c_{i-1})$ from the proper-orientedness of $\mathcal{R}$ (or $\mathcal{S}$). Thus, $\mathcal{V}(l) \cup \mathcal{V}(c_k) = \mathcal{V}(l) \cup \mathcal{V}(c_{k-1})$. Hence, $\rho_k := \rho_{k-1}$ satisfies (a)–(c). Suppose otherwise. Then t_k is linear and is a constructor term w.r.t. both $\mathcal{R}$ and $\mathcal{S}$. Then, by $t_k\theta \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} s'$, there exists a substitution ρ such that $s' = t_k\rho$ and $dom(\rho) \subseteq \mathcal{V}(t_k)$ such that for any $x \in \mathcal{V}(t_k)$, $x\theta \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} x\rho$. Furthermore, by $t_k\theta' \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} s'$, there exists a substitution ρ' such that $s' = t_k\rho'$ and $dom(\rho') \subseteq \mathcal{V}(t_k)$ such that for any $x \in \mathcal{V}(t_k)$, $x\theta' \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} x\rho'$. Now, because $t_k\rho = s' = t_k\rho'$, we know $x\rho = x\rho'$ for any $x \in \mathcal{V}(t_k)$, and thus $\rho = \rho'$ from $dom(\rho), dom(\rho') \subseteq \mathcal{V}(t_k)$. We also have $\mathcal{V}(t_k) \cap (\mathcal{V}(l) \cup \mathcal{V}(c_{k-1})) = \emptyset$ by the right-stability of $\mathcal{R}$ (or $\mathcal{S}$), and thus, $dom(\rho) \cap dom(\rho_{k-1}) = \emptyset$. Hence, $\rho_k := \rho_{k-1} \cup \rho$ is a substitution, and ρ_k satisfies (a)–(c). This completes the induction proof for existence of substitutions ρ_k satisfying (a)–(c) ($1 \leq k \leq j$).

Now consider the substitution ρ_j. Since $\mathcal{R}$ (and $\mathcal{S}$) is a 3-CTRS, we have $\mathcal{V}(r) \subseteq \mathcal{V}(l) \cup \mathcal{V}(c_j)$. Thus, by the condition (c), $N_i = r\theta \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} r\rho_j$ and $P_i = r\theta' \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} r\rho_j$ hold. Thus, taking $Q_i := r\rho_j$, and we have $N_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{n-1}} Q_i$ and $P_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_{m-1}} Q_i$.

ii Case (β). Let $M_i \overset{A'_1}{\rightarrow}_{\mathcal{R}_m} \tilde{M}_i \overset{A'_2,\ldots,A'_{\bar{q}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_n} N_i$. The proof of this case is illustrated in Fig. 3 (left). Because there exists an overlap between $l \rightarrow r \Leftarrow c \in \mathcal{R}$ and $l' \rightarrow r' \Leftarrow c' \in \mathcal{S}$, there is substitution θ'' and a position $p \in \mathrm{Pos}_{\mathcal{F}}(l')$ such that $M_i = l'\theta'' = l'\theta''[l\theta'']_p = l\theta''[A'_1]_p$. Then, $\tilde{M}_i = l'[r]_p\theta''$, $P_i = r'\theta''$, $\mathcal{R}_{m-1} \vdash c\theta''$ and $\mathcal{S}_{n-1} \vdash c'\theta''$. Then, there exists an CCP $\langle u, v\rangle \Leftarrow \langle d, d'\rangle \in CCP(\mathcal{R}, \mathcal{S})$, where $u = l'[r]_p\sigma$, $v = r'\sigma$, $d = c\sigma$ and $d' = c'\sigma$ for the mgu σ of $l'|_p$ and l. Then, as $(l'\theta'')_p = l\theta''$, we have $\theta'' = \rho \circ \sigma$ for some ρ. Thus, $P_i = r'\theta'' = (r'\sigma)\rho = v\rho$, $\tilde{M}_i = l'[r]_p\theta'' = (l'[r]_p\sigma)\rho = u\rho$, $\mathcal{R}_{m-1} \vdash d\rho$, and $\mathcal{S}_{n-1} \vdash d'\rho$. Hence, by our critical pair condition (2), $u\rho \hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow_{\mathcal{S}_n} \circ \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{<n}} s$ and $v\rho \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_m} s$ for some s, and thus, by taking $\tilde{P}_i := s\rho$, we have $\tilde{M}_i \hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow_{\mathcal{S}_n} \tilde{P}'_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_{<n}} \tilde{P}_i$ and $P_i \overset{*}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{R}_m} \tilde{P}_i$ for some $\tilde{P}'_i$.

Suppose $\tilde{M}_i \overset{C_1,\ldots,C_{\bar{r}}}{\hookrightarrow\!\!\!\!\!+\!\!\!\!\rightarrow}_{\mathcal{S}_n} \tilde{P}'_i$. Let $\Gamma' = \{A'_i \mid \exists C_j.A'_i \subset C_j\} \cup \{C_i \mid \exists A'_j.C_i \subseteq A'_j\}$. Occurrences in Γ' are distinct, and for any $\tilde{C} \in \Gamma'$, there exists A'_j ($2 \leq j \leq \bar{q}$) such that $\tilde{C} \subseteq A'_j$. Hence, $|\Gamma'| \leq$

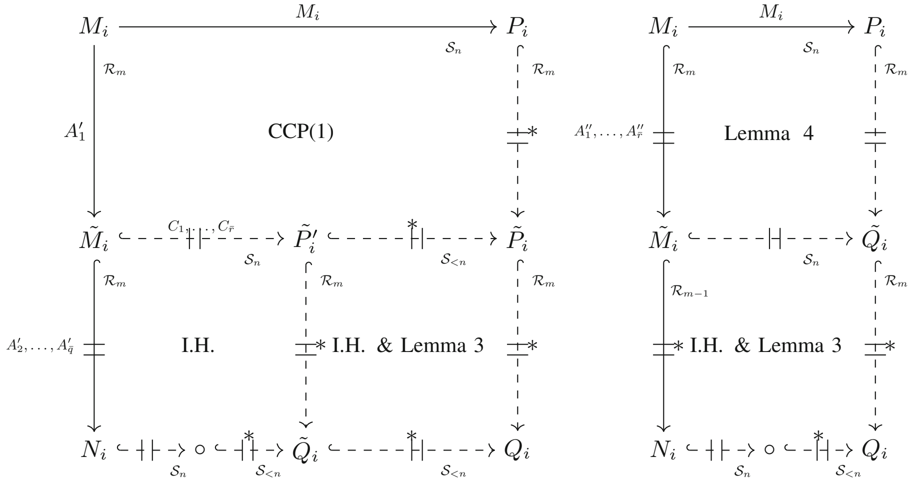


Fig. 3. Case 2.(a).ii (left) and Case 2.(b) (right)

$\sum_{j=2}^{\bar{q}} |A'_j| < \sum_{j=1}^{\bar{q}} |A'_j| \leq |\Gamma|$. Thus, one can apply induction hypothesis to obtain $\tilde{Q}_i$ such that $N_i \mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{S}_{<n}} \tilde{Q}_i$ and $\tilde{P}'_i \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{R}_m} \tilde{Q}_i$. By applying induction hypothesis and Lemma 3 once again, we know that there exists Q_i such that $\tilde{Q} \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{S}_{<n}} Q_i \overset{*}{\mathrel{\leftarrow\!\!\!\Vert\!\!\!\hookleftarrow}}_{\mathcal{R}_m} \tilde{P}_i$.

(b) Case that $A'_j \xrightarrow{*}_{\mathcal{R}_{m-1}} \tilde{A}'_j$ for any $A'_j \in X'$. Since $M_i \overset{A'_1,\dots,A'_{\bar{q}}}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{R}_m} N_i$ and $A'_1, \dots, A'_{\bar{q}}$ are parallel, one can rewrite $A'_j \in Y'$ first. That is, $M_i \overset{A''_1,\dots,A''_{\bar{r}}}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{R}_m} \tilde{M}_i \xrightarrow{*}_{\mathcal{R}_{m-1}} N_i$ where $Y' = \{A''_1, \dots, A''_{\bar{r}}\}$. The proof of this case is illustrated in Fig. 3 (right). Then, as each A''_j is contained in θ', by Lemma 4, there exists $\tilde{Q}$ such that $\tilde{M}_i \rightarrow_{\mathcal{S}_n} \tilde{Q}$ and $P_i \mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}_{\mathcal{R}_m} \tilde{Q}$. Furthermore, as $\rightarrow_{\mathcal{S}_n} \subseteq \mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}_{\mathcal{S}_n}$ and $\xrightarrow{*}_{\mathcal{R}_{m-1}} \subseteq \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{R}_{m-1}}$, one can apply induction hypothesis and Lemma 3 to $N_i \xleftarrow{*}_{\mathcal{R}_{m-1}} \tilde{M}_i \rightarrow_{\mathcal{S}_n} \tilde{Q}$ to obtain Q_i such that $N_i \mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}_{\mathcal{S}_n} \circ \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{S}_{<n}} Q_i$ and $\tilde{Q} \overset{*}{\mathrel{\hookrightarrow\!\!\!\!\Vert\!\!\rightarrow}}_{\mathcal{R}_m} Q_i$.

Finally, from Lemma 2 we conclude that $\mathcal{R}$ and $\mathcal{S}$ are level-commutative. □

A level-confluence criterion is obtained by taking $\mathcal{R} = \mathcal{S}$. Note that one can use CCP_{out} instead of CCP in the first condition, contrast to the commutativity criterion, as the second condition implies the part for $CCP_{in}(\mathcal{R})$ of it.

Corollary 1. *Let $\mathcal{R}$ be a left-linear, properly oriented, right-stable 3-CTRS. If the following conditions are satisfied, then $\mathcal{R}$ is level-confluent:*

1. *for any $\langle u, v\rangle \Leftarrow \langle c, c'\rangle \in CCP_{out}(\mathcal{R})$, $m, n \geq 1$ and substitution ρ, if $c\rho \subseteq \xrightarrow{*}_{\mathcal{R}_{m-1}}$ and $c'\rho \subseteq \xrightarrow{*}_{\mathcal{R}_{n-1}}$ then $u\rho \mathrel{-\!\!\!\Vert\!\!\rightarrow}_{\mathcal{R}_n} \circ \xrightarrow{*}_{\mathcal{R}_{<n}} \circ \xleftarrow{*}_{\mathcal{R}_m} v\rho$, and*
2. *for any $\langle v, u\rangle \Leftarrow \langle c', c\rangle \in CCP_{in}(\mathcal{R})$, $m, n \geq 1$ and substitution ρ, if $c\rho \subseteq \xrightarrow{*}_{\mathcal{R}_{m-1}}$ and $c'\rho \subseteq \xrightarrow{*}_{\mathcal{R}_{n-1}}$ then $v\rho \mathrel{-\!\!\!\Vert\!\!\rightarrow}_{\mathcal{R}_m} \circ \xrightarrow{*}_{\mathcal{R}_{<m}} u\rho$.*

Example 1. Let $\mathcal{R}$ and $\mathcal{S}$ be the following CTRSs:

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{p}(x) \to \mathsf{q}(x) & \\ \mathsf{r}(x) \to \mathsf{s}(\mathsf{p}(x)) & \\ \mathsf{s}(x) \to \mathsf{f}(y) & \Leftarrow \mathsf{p}(x) \approx y \end{array} \right\} \quad \mathcal{S} = \left\{ \begin{array}{ll} \mathsf{p}(x) \to \mathsf{r}(x) & \\ \mathsf{q}(x) \to \mathsf{s}(\mathsf{p}(x)) & \\ \mathsf{s}(x) \to \mathsf{f}(y) & \Leftarrow \mathsf{p}(x) \approx y \end{array} \right\}$$

We have $CCP(\mathcal{R}, \mathcal{S}) = \{\langle \mathsf{q}(x), \mathsf{r}(x) \rangle \Leftarrow \langle \emptyset, \emptyset \rangle\}$ and $CCP_{in}(\mathcal{S}, \mathcal{R}) = \emptyset$. Note that the overlap of $\mathsf{s}(x) \to \mathsf{f}(y) \Leftarrow p(x) \approx y \in \mathcal{R}$ and $\mathsf{s}(x) \to \mathsf{f}(y) \Leftarrow \mathsf{p}(x) \approx y \in \mathcal{S}$ is not considered, as these rules are identical; the case 2.(a).i of the proof above treats this case. Now, because we have $\mathsf{q}(x) \to_{\mathcal{S}_n} \mathsf{s}(\mathsf{p}(x))$ and $\mathsf{r}(x) \to_{\mathcal{R}_m} \mathsf{s}(\mathsf{p}(x))$ $(n, m \geq 1)$ the condition (1) of the Theorem 1 is satisfied. Other conditions of the theorem are also satisfied. Thus, $\mathcal{R}$ and $\mathcal{S}$ are level-commutative. Similarly, one can show $\mathcal{R} \cup \mathcal{S}$ is level-confluent.

Example 2. Take CTRSs $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}_f$ and $\mathcal{S} = \mathcal{S}' \cup \mathcal{R}_f$ such that

$$\mathcal{R}' = \left\{ \begin{array}{l} \mathsf{p}(x, y) \to \mathsf{r}(x, y) \Leftarrow x \approx \mathsf{a} \\ \mathsf{q}(x, y) \to \mathsf{p}(x, y) \Leftarrow x \approx \mathsf{a} \end{array} \right\} \quad \mathcal{S}' = \left\{ \begin{array}{l} \mathsf{p}(x, y) \to \mathsf{q}(x, y) \Leftarrow y \approx \mathsf{b} \\ \mathsf{r}(x, y) \to \mathsf{p}(x, y) \Leftarrow y \approx \mathsf{b} \end{array} \right\}$$

and $\mathcal{R}_f = \{\mathsf{f}(0) \to \mathsf{a},\ \mathsf{f}(\mathsf{s}(x)) \to \mathsf{b} \Leftarrow \mathsf{f}(x) \approx \mathsf{a},\ \mathsf{f}(\mathsf{s}(x)) \to \mathsf{a} \Leftarrow \mathsf{f}(x) \approx \mathsf{b}\}$. We have $CCP(\mathcal{R}, \mathcal{S}) = \{\ (a) : \langle \mathsf{r}(x, y), \mathsf{q}(x, y) \rangle \Leftarrow \langle \{x \approx \mathsf{a}\}, \{y \approx \mathsf{b}\} \rangle,\ (b) : \langle \mathsf{a}, \mathsf{b} \rangle \Leftarrow \langle \{\mathsf{f}(x) \approx \mathsf{b}\}, \{\mathsf{f}(x) \approx \mathsf{a}\} \rangle,\ (c) : \langle \mathsf{b}, \mathsf{a} \rangle \Leftarrow \langle \{\mathsf{f}(x) \approx \mathsf{a}\}, \{\mathsf{f}(x) \approx \mathsf{b}\} \rangle\}$, and $CCP_{in}(\mathcal{S}, \mathcal{R}) = \emptyset$. For the CCP (a), let $m, n \geq 1$ and ρ be any substitution, and suppose that $\rho(x) \to_{\mathcal{R}_{m-1}} \mathsf{a}$ and $\rho(y) \to_{\mathcal{S}_{n-1}} \mathsf{b}$. Then, we have $\mathsf{r}(\rho(x), \rho(y)) \to_{\mathcal{S}_n} \mathsf{p}(\rho(x), \rho(y))$ and $\mathsf{q}(\rho(x), \rho(y)) \to_{\mathcal{R}_m} \mathsf{p}(\rho(x), \rho(y))$. Also, note that there is no term t such that $t \xrightarrow{*}_{\mathcal{R}} \mathsf{b}$ and $t \xrightarrow{*}_{\mathcal{S}} \mathsf{a}$ (or $t \xrightarrow{*}_{\mathcal{R}} \mathsf{a}$ and $t \xrightarrow{*}_{\mathcal{S}} \mathsf{b}$). Thus, the condition (1) of the Theorem 1 holds for CCPs (a)–(c). Other conditions of the theorem are also satisfied. Thus, $\mathcal{R}$ and $\mathcal{S}$ are level-commutative. Similarly, one can show $\mathcal{R} \cup \mathcal{S}$ is level-confluent.

Since TRSs can be regarded as CTRSs with no conditions and they are trivially properly-oriented, right-stable, and of type 3, this theorem covers Proposition 1. However, this does not mean our theorem broaden the scope of TRSs that can be guaranteed to commute—because rewrite steps of TRSs are level 1 rewrite steps in CTRSs, our condition reduces to the one of Proposition 1 in TRSs. Thus, when restricting to TRSs, Theorem 1 coincides Proposition 1.

On the other hand, Corollary 1 properly extends Proposition 2, as witnessed by $\mathcal{R} \cup \mathcal{S}$ in Examples 1, 2.

4 Critical Pair Criteria for Join and Semi-Equational CTRSs

In this section, we explore critical pair criteria for join and semi-equational CTRSs, following our approach in the previous section.

First, let us fix additional notions and notations that will be used in this section. A rewrite step of *join* CTRS $\mathcal{R}$ is defined via the following TRS $\mathcal{R}_n$

($n \in \mathbb{N}$), which are inductively given as follows: $\mathcal{R}_0 = \emptyset$, $\mathcal{R}_{n+1} = \{l\sigma \to r\sigma \mid l \to r \Leftarrow c \in \mathcal{R}, c\sigma \subseteq \xrightarrow{*}_{\mathcal{R}_n} \circ \xleftarrow{*}_{\mathcal{R}_n}\}$. For *semi-equational* CTRS $\mathcal{R}$, we modify the second clause as: $\mathcal{R}_{n+1} = \{l\sigma \to r\sigma \mid l \to r \Leftarrow c \in \mathcal{R}, c\sigma \subseteq \xleftrightarrow{*}_{\mathcal{R}_n}\}$. Similarly to the oriented case, a rewrite step $s \to_{\mathcal{R}} t$ of $\mathcal{R}$ is given as $s \to_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}_n} t$ for some n, and the smallest n such that $s \to_{\mathcal{R}_n} t$ is called the *level* of the rewrite step $s \to_{\mathcal{R}} t$. We write $\downarrow_{\mathcal{R}_n}$ ($\downarrow_{\mathcal{R}}$) for the relation $\xrightarrow{*}_{\mathcal{R}_n} \circ \xleftarrow{*}_{\mathcal{R}_n}$ (resp. $\xrightarrow{*}_{\mathcal{R}} \circ \xleftarrow{*}_{\mathcal{R}}$).

In this section (except Subsect. 4.2), in order to distinguish three types of CTRSs, we write $\mathcal{R}^{\mathsf{o}}$ for an oriented CTRS, $\mathcal{R}^{\mathsf{j}}$ for a join CTRS, and $\mathcal{R}^{\mathsf{s}}$ for a semi-equational CTRS. Similarly, notations $\mathcal{R}^{\mathsf{o}}_n, \mathcal{R}^{\mathsf{j}}_n, \dots$ are employed. Notations $\mathcal{R}^{\mathsf{o}}_n \vdash c\sigma$ ($\mathcal{R}^{\mathsf{j}}_n \vdash c\sigma$, $\mathcal{R}^{\mathsf{s}}_n \vdash c\sigma$) stands for $c\sigma \subseteq \xrightarrow{*}_{\mathcal{R}^{\mathsf{o}}_n}$ (resp. $c\sigma \subseteq \downarrow_{\mathcal{R}^{\mathsf{j}}_n}$, $c\sigma \subseteq \xleftrightarrow{*}_{\mathcal{R}^{\mathsf{s}}_n}$).

The following basic relations between rewrite relation on three types of CTRSs on each level are essentially proved in [18, Lemmas 1 and 2].

Lemma 5. *Let $\mathcal{R}$ be a CTRS. Then $\to_{\mathcal{R}^{\mathsf{o}}_n} \subseteq \to_{\mathcal{R}^{\mathsf{j}}_n} \subseteq \to_{\mathcal{R}^{\mathsf{s}}_n}$ for each n.*

Notions of orthogonality, proper-orientedness and right-stability are syntax-oriented, and their definitions remain same for other types of CTRSs. Note that even under the conditions of proper-orientedness and right-stability, $\to_{\mathcal{R}^{\mathsf{o}}_n} = \to_{\mathcal{R}^{\mathsf{j}}_n}$ does not hold in general.

4.1 Level-Confluence of Join and Semi-Equational 3-CTRSs

In [14, Corollary 5.3], Proposition 2 is applied to show the corresponding class of join CTRSs are level-confluent:

Proposition 3 ([14]). *Let $\mathcal{R}$ be an orthogonal, properly oriented, right-stable 3-CTRS. Then $\mathcal{R}^{\mathsf{j}}$ is level-confluent.*

Given our Theorem 1, a natural question is whether a similar extension is possible for our theorem. In this subsection, we give a partially positive answer to this question—we generalize the result above to the level-confluence part (Corollary 1) of our theorem, even though a similar extension does not work for level-commutation. Indeed, we show that above proposition can be extended to a more general setting of CTRSs where the orthogonality requirement is replaced with level-confluence of $\mathcal{R}^{\mathsf{o}}$. Furthermore, the generalization is obtained not only for join CTRSs but also for semi-equational CTRSs.

The next two lemmas are abstractions of the ones [14, Lemmas 5.1 and 5.2], where the proofs remain almost the same.

Lemma 6. *Let $\mathcal{R}$ be a properly oriented, right-stable 3-CTRS such that $\mathcal{R}^{\mathsf{o}}$ is level-confluent. Let $l \to r \Leftarrow s_1 \approx t_1, \dots, s_j \approx t_j \in \mathcal{R}$. If $s_i\sigma \downarrow_{\mathcal{R}^{\mathsf{o}}_{n-1}} t_i\sigma$ for any $1 \le i \le j$ then $l\sigma \downarrow_{\mathcal{R}^{\mathsf{o}}_n} r\sigma$.*

Lemma 7. *Let $\mathcal{R}$ be a properly oriented, right-stable 3-CTRS such that $\mathcal{R}^{\mathsf{o}}$ is level-confluent. If $s \to_{\mathcal{R}^{\mathsf{s}}_n} t$ then $s \downarrow_{\mathcal{R}^{\mathsf{o}}_n} t$.*

Now we present the claimed result:

Theorem 2. *Let $\mathcal{R}$ be a properly oriented, right-stable 3-CTRS. If $\mathcal{R}^{\mathrm{o}}$ is level-confluent then $\mathcal{R}^{\mathrm{j}}$ and $\mathcal{R}^{\mathrm{s}}$ are level-confluent.*

Proof. Let $\mathcal{R}$ be a properly oriented, right-stable 3-CTRS such that $\mathcal{R}^{\mathrm{o}}$ is level-confluent. Suppose $t_1 \xleftarrow{*}_{\mathcal{R}^{\mathrm{j}}_n} s \xrightarrow{*}_{\mathcal{R}^{\mathrm{j}}_n} t_2$ ($t_1 \xleftarrow{*}_{\mathcal{R}^{\mathrm{s}}_n} s \xrightarrow{*}_{\mathcal{R}^{\mathrm{s}}_n} t_2$). Then $t_1 \xleftarrow{*}_{\mathcal{R}^{\mathrm{s}}_n} s \xrightarrow{*}_{\mathcal{R}^{\mathrm{s}}_n} t_2$ by Lemma 5. Thus, by Lemma 7, $t_1 \xleftrightarrow{*}_{\mathcal{R}^{\mathrm{o}}_n} t_2$. Hence, $t_1 \downarrow_{\mathcal{R}^{\mathrm{o}}_n} t_2$ follows by the level-confluence of $\mathcal{R}^{\mathrm{o}}$. Using again Lemma 5, this implies $t_1 \downarrow_{\mathcal{R}^{\mathrm{j}}_n} t_2$ (resp. $t_1 \downarrow_{\mathcal{R}^{\mathrm{s}}_n} t_2$). □

Thus, Corollary 1 can be applied to show the level-confluence of join and semi-equational CTRSs. Note here that the conditions of Corollary 1 is stated in terms of $\to^{\mathrm{o}}_{\mathcal{R}}$ not in that of $\to^{\mathrm{j}}_{\mathcal{R}}$ or $\to^{\mathrm{s}}_{\mathcal{R}}$.

4.2 Commutation of Semi-Equational 3-CTRSs

A most fundamental ingredient of the proof presented (inherited from [14]) is to use induction on the level of rewrite relation. It seems, however, applying this approach for join and semi-equational CTRSs contains fundamental difficulty. Without the induction on the level, what can we do within the parallel-closed approach? In this subsection, we will exhibit one alternative approach for semi-equational CTRSs.

In [1], it is reported that left-linear parallel-closed semi-equational 1-CTRSs are confluent. By examining its proof detail, we can extend it to commutativity of 3-CTRSs as follows. Below, notation $\mathcal{R} \vdash c\sigma$ (etc.) stands for $c\sigma \subseteq \xleftrightarrow{*}_{\mathcal{R}}$.

Theorem 3. *Let $\mathcal{R}, \mathcal{S}$ be semi-equational left-linear 3-CTRSs. Suppose the following conditions are satisfied:*

1. *for any $\langle u, v\rangle \Leftarrow \langle c, c'\rangle \in CCP(\mathcal{R}, \mathcal{S})$ and any substitution ρ, if $\mathcal{R} \vdash c\sigma$ and $\mathcal{S} \vdash c'\sigma$, then $u\rho \mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{S}} \circ \xleftarrow{*}_{\mathcal{R}} v\rho$, and*
2. *for any $\langle v, u\rangle \Leftarrow \langle c', c\rangle \in CCP_{in}(\mathcal{S}, \mathcal{R})$ and any substitution ρ, if $\mathcal{R} \vdash c\rho$ and $\mathcal{S} \vdash c'\rho$, then $v\rho \mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{R}} u\rho$.*

Furthermore, assume $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{S}} \subseteq \xleftrightarrow{*}_{\mathcal{R}}$, $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{R}} \subseteq \xleftrightarrow{*}_{\mathcal{S}}$ and $\mathcal{R} \cap \mathcal{S}$ is a 2-CTRS. Then, $\mathcal{R}$ and $\mathcal{S}$ commute.

We remark that conditions $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{S}} \subseteq \xleftrightarrow{*}_{\mathcal{R}}$ and $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{R}} \subseteq \xleftrightarrow{*}_{\mathcal{S}}$ are used to close nested peaks, and that the condition that $\mathcal{R} \cap \mathcal{S}$ is a 2-CTRS is required to resolve for peaks obtained by the same rule.

Example 3. Let $\mathcal{R}$ and $\mathcal{S}$ be the following left-linear semi-equational 3-CTRSs:

$$\mathcal{R} = \{\mathsf{q}(x,y) \to \mathsf{p}(y,x),\ \mathsf{p}(x,y) \to \mathsf{q}(x',y') \Leftarrow x \approx x', y \approx y'\}$$
$$\mathcal{S} = \{\mathsf{p}(x,y) \to \mathsf{q}(y,x),\ \mathsf{q}(x,y) \to \mathsf{p}(x',y') \Leftarrow x \approx x', y \approx y'\}$$

By induction on the level n, one can show $\to_{\mathcal{S}_n} \subseteq \xrightarrow{*}_{\mathcal{R}_n}$ and $\to_{\mathcal{R}_n} \subseteq \xrightarrow{*}_{\mathcal{S}_n}$. Thus, conditions $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{S}} \subseteq \xleftrightarrow{*}_{\mathcal{R}}$ are $\mathrel{+\!\!\!\!\!\!\longrightarrow}_{\mathcal{R}} \subseteq \xleftrightarrow{*}_{\mathcal{S}}$ are satisfied. Clearly, $\mathcal{R} \cap \mathcal{S} = \emptyset$ is a 2-CTRS. We have $CCP(\mathcal{R}, \mathcal{S}) = \{\langle \mathsf{q}(x',y'), \mathsf{q}(y,x)\rangle \Leftarrow \langle\{x \approx x', y \approx$

$y'\}, \emptyset\rangle, \{\langle \mathsf{p}(y,x), \mathsf{p}(x',y')\rangle \Leftarrow \langle \emptyset, \{x \approx x', y \approx y'\}\rangle\}$ and $CCP_{in}(\mathcal{S},\mathcal{R}) = \emptyset$. Clearly, $\rho(x) \overset{*}{\leftrightarrow}_{\mathcal{R}} \rho(x')$ and $\rho(y) \overset{*}{\leftrightarrow}_{\mathcal{R}} \rho(y')$ imply $\mathsf{p}(\rho(x'),\rho(y')) \to_{\mathcal{R}} \mathsf{q}(\rho(y),\rho(x))$, and $\rho(x) \overset{*}{\leftrightarrow}_{\mathcal{S}} \rho(x')$ and $\rho(y) \overset{*}{\leftrightarrow}_{\mathcal{S}} \rho(y')$ imply $\mathsf{q}(\rho(x),\rho(y)) \leftarrow_{\mathcal{S}} \mathsf{p}(\rho(y),\rho(x))$. Thus, all conditions of the Theorem 3 are satisfied. Thus, $\mathcal{R}$ and $\mathcal{S}$ commute.

Note the conditions $\twoheadrightarrow\!\!\!\!\!\!\shortmid\shortmid_{\mathcal{S}} \subseteq \overset{*}{\leftrightarrow}_{\mathcal{R}}$ and $\twoheadrightarrow\!\!\!\!\!\!\shortmid\shortmid_{\mathcal{R}} \subseteq \overset{*}{\leftrightarrow}_{\mathcal{S}}$ of Theorem 3 imply $\overset{*}{\leftrightarrow}_{\mathcal{R}} = \overset{*}{\leftrightarrow}_{\mathcal{S}}$, i.e. $\mathcal{R}$ and $\mathcal{S}$ have the same underlying logic.

5 Conclusion

We have given a critical pair criterion for ensuring level-commutativity of left-linear properly-oriented right-stable oriented 3-CTRSs. Our result generalizes a sufficient criterion for commutativity of left-linear TRSs of Toyama [16]. It also properly extends level-confluence of orthogonal properly-oriented right-stable oriented 3-CTRSs of Suzuki et al. [14]. We then have showed this result can be applied to obtain a criterion for level-confluence of left-linear properly-oriented right-stable join and semi-equational 3-CTRSs, generalizing a result of [14]. We have also explored a similar but different approach of Aoto and Toyama [1] to obtain a criterion for the commutation of semi-equational 3-CTRSs.

Wirth [17] also gave a criterion of level-confluence for possibly non-orthogonal CTRSs that generalizes a sufficient criterion for confluence of left-linear TRSs of [16]. He adapted the approach of [16] for a framework of join CTRSs. It also incorporates some ideas of [14] so as to give the notions of (weak-)quasi-normal CTRSs, etc. A critical key difference with the usual conditional rewriting such as employed in our paper, however, is that the validity of conditions needs to be satisfied under a kind of constructor discipline. This restriction considerably simplifies proof arguments dealing with conditional parts, paying the penalty of going apart from the standard framework. On the other hand, despite these sharp differences on the underlying frameworks of ours and [17], interestingly, the critical pair criterion of Theorem 3 and Wirth's critical pair criterion ([17, Definition 28]) resemble very much.

Over various formalisms of rewriting, considerable efforts have been spent on automating confluence checks in recent years. Yearly competition[2] of confluence tools started in 2012; the category of CTRS has been also introduced in 2014. In recent competitions, confluence of *oriented 3-CTRSs*, which our main theorem deal with, has been focused in the category of CTRS. Known confluence tools for CTRSs include CONFident [6], ConCon [13], CO3 [9] and ACP [2]. We note here that all these tools fail to show confluence of $\mathcal{R} \cup \mathcal{S}$ of Example 2[3]. Among these tools (at least) ConCon and ACP incorporate checking of confluence criterion of [14]. We have been working on the automation of our results, but it is yet under

[2] http://project-coco.uibk.ac.at/.

[3] Experimented for CoCo 2022 participants ACP, CO3, CONFident and a CoCo 2020 participant ConCon, via CoCoWeb [7].

development. Recent advances in confluence tools for CTRSs include automation of infeasibility checking [5]—we believe some approaches for automation of infeasibility checking can be adapted for automation of our criterion.

Formalization by interactive theorem provers such as Isabelle/HOL, Coq, PVS4, etc. have been of great interest in recent years. Formalization is also indispensable for certification of results obtained by confluence tools. Regarding for results of [14], a formalization in Isabelle/HOL has been reported by Sternagel and Sternagel [12]. On the other hand, formalization of our results remains completely as a future work.

Acknowledgements. Thanks are due to the anonymous reviewers (including those for all previous versions of the paper) for valuable comments. This work is partially supported by JSPS KAKENHI No. 21K11750.

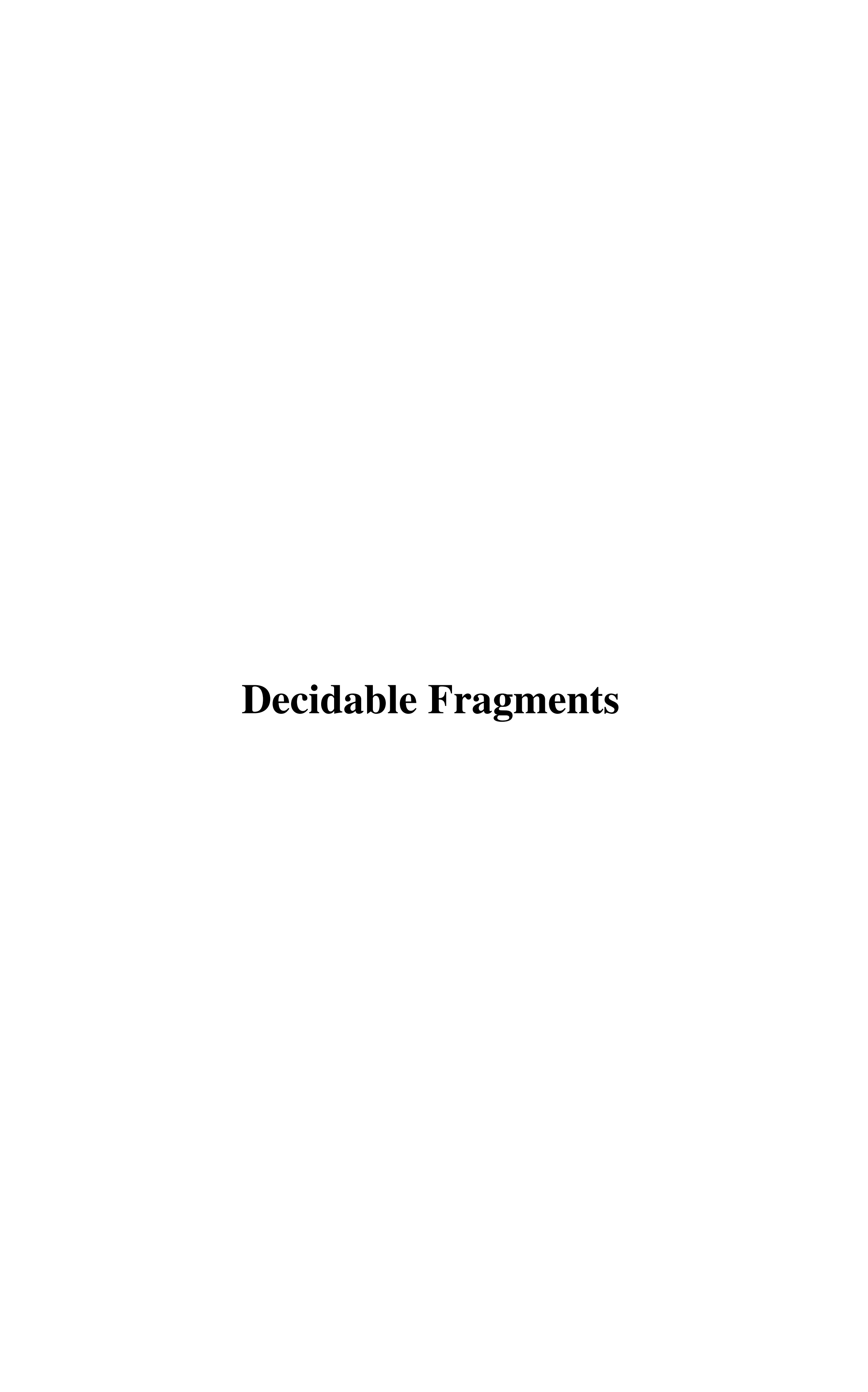
References

1. Aoto, T., Toyama, Y.: Automated proofs of unique normal forms w.r.t. conversion for term rewriting systems. In: Herzig, A., Popescu, A. (eds.) FroCoS 2019. LNCS (LNAI), vol. 11715, pp. 330–347. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29007-8_19
2. Aoto, T., Yoshida, J., Toyama, Y.: Proving confluence of term rewriting systems automatically. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 93–102. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_7
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
4. Gramlich, B.: Confluence without termination via parallel critical pairs. In: Kirchner, H. (ed.) CAAP 1996. LNCS, vol. 1059, pp. 211–225. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61064-2_39
5. Gutiérrez, R., Lucas, S.: Automatically proving and disproving feasibility conditions. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 416–435. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_27
6. Gutiérrez, R., Lucas, S., Vítores, M.: Confluence of conditional rewriting in logic form. In: Proceedings of 41st FSTTCS. LIPIcs, vol. 213, pp. 44:1–44:18. Schloss Dagstuhl (2021)
7. Hirokawa, N., Nagele, J., Middeldorp, A.: Cops and CoCoWeb: infrastructure for confluence tools. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 346–353. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_23
8. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems. J. ACM **27**(4), 797–821 (1980)
9. Nishida, N., Kuroda, T., Yanagisawa, M., Gmeiner, K.: CO3: a COnverter for proving COnfluence of COnditional TRSs (version 1.2). In: Proceedings of 4th IWC (2015). https://www.trs.cm.is.nagoya-u.ac.jp/co3/papers/co3_2015_full.pdf
10. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3661-8
11. Okui, S.: Simultaneous critical pairs and Church-Rosser property. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 2–16. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052357

12. Sternagel, C., Sternagel, T.: Level-confluence of 3-CTRSs in Isabelle/HOL. In: Proceedings of 4th IWC, pp. 28–32 (2015)
13. Sternagel, T., Middeldorp, A.: Conditional confluence (system description). In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 456–465. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_31
14. Suzuki, T., Middeldorp, A., Ida, T.: Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914, pp. 179–193. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59200-8_56
15. Terese: Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
16. Toyama, Y.: Commutativity of term rewriting systems. In: Programming of Future Generation Computer II, pp. 393–407. North-Holland, Amsterdam (1987)
17. Wirth, C.P.: Shallow confluence of conditional term rewriting systems. J. Symb. Comput. **44**, 60–98 (2009)
18. Yamada, T., Avenhaus, J., Loría-Sáenz, C., Middeldorp, A.: Logicality of conditional rewrite systems. Theoret. Comput. Sci. **236**, 209–232 (2000)
19. Yoshida, J., Aoto, T., Toyama, Y.: Automating confluence check of term rewriting systems. Comput. Softw. **26**(2), 76–92 (2009). in Japanese

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.



Decidable Fragments

Logic of Communication Interpretation: How to Not Get Lost in Translation

Giorgio Cignarale, Roman Kuznets, Hugo Rincon Galeana(✉), and Ulrich Schmid

TU Wien, Vienna, Austria
{giorgio.cignarale,roman.kuznets,hugo.galeana}tuwien.ac.at, s@ecs.tuwien.ac.at

Abstract. Byzantine fault-tolerant distributed systems are designed to provide resiliency despite arbitrary faults, i.e., even in the presence of agents who do not follow the common protocol and/or despite compromised communication. It is, therefore, common to focus on the perspective of correct agents, to the point that the epistemic state of byzantine agents is completely ignored. Since this view relies on the assumption that faulty agents may behave arbitrarily adversarially, it is overly conservative in many cases. In blockchain settings, for example, dishonest players are usually not malicious, but rather selfish, and thus just follow some "hidden" protocol that is different from the protocol of the honest players. Similarly, in high-availability large-scale distributed systems, software updates cannot be globally instantaneous, but are rather performed node-by-node. Consequently, updated and non-updated nodes may simultaneously be involved in a protocol for solving a distributed task like consensus or transaction commit. Clearly, the usual assumption of common knowledge of the protocol is inappropriate in such a setting. On the other hand, joint protocol execution and, sometimes, even basic communication becomes problematic without this assumption: How are agents supposed to interpret each other's messages without knowing their mutual communication protocols? We propose a novel epistemic modality *creed* for epistemic reasoning in heterogeneous distributed systems with agents that are uncertain of the actual communication protocol used by their peers. We show that the resulting logic is quite closely related to modal logic S5, the standard logic of epistemic reasoning in distributed systems. We demonstrate the utility of our approach by several examples.

1 Introduction

A *distributed system* is a system with multiple processes, or agents, located on different machines that communicate and coordinate actions, via *message*

G. Cignarale and R. Kuznets—Supported by the Austrian Science Fund (FWF) projects ByzDEL (P33600).

H.R. Galeana—Supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

© The Author(s) 2023

U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 119–136, 2023.
https://doi.org/10.1007/978-3-031-43369-6_7

passing or *shared memory*, in order to accomplish some task [8,21]. This common task is achieved by means of agent protocols instructing agents how to exchange information and act. Designing distributed systems is difficult due to the inherent uncertainty agents have about the global state of the system, caused, e.g., by different computation speeds and message delays.

Knowledge [15] is a powerful conceptual way of reasoning about this uncertainty [13,14]. Indeed, knowledge is at the core of the agents' ability to act according to the protocol: According to the Knowledge of Preconditions principle [22], a protocol instruction to act based on a precondition φ can only be followed if the agent knows φ to hold. While trivial for preconditions based on the local state of the acting agent itself, this observation comes to the fore for global preconditions, also involving other agents, as is common for coordination problems such as consensus.

One of the standard ways of modeling agents' knowledge is via the possible world semantics that takes into account all the possible global states the agents can be in and which of these possible worlds a particular agent can distinguish based on its local information. In this view, agent i knows a proposition φ, written $K_i\varphi$, in a global state s iff this proposition holds in all global states s' that are indistinguishable from s for i. The primary means of obtaining new knowledge — and the only way of increasing knowledge about the local states of other agents — in a distributed system is by means of communication.

Fault-tolerant systems add another layer of complexity, in particular, when processes may not only stop operating or drop messages but can be (or become) *byzantine* [19], i.e., may behave arbitrarily erroneously, in particular, can communicate in erratic, arbitrary, or deceptive manner. Malicious faulty agents may have a "hidden agenda", in which case, instead of following the original commonly known protocol, a faulty agent (or a group of faulty agents) can execute actions (possibly in consort with each other) that jeopardize the original goals of the system.

Although these hidden agendas are typically not transparent for correct agents, some assumptions must be made to restrict the types and numbers of protocol-defying actions and messages. Without such restrictions, provably correct solutions for a distributed task do not exist. These assumptions must usually be commonly known by all agents, like the basic communication mechanism, the protocol of all correct agents, the data encoding used in its messages, etc. In [7], the whole corpus of these common assumptions is referred to as *a priori knowledge*.[1] For the possible world semantics, this translates into the assumption of common knowledge of the model [3], which enables the agents to compute epistemic states of other agents, a task necessary for a typical coordination problem like consensus [6].

Since correct agents generally cannot distinguish a simple malfunction from malintent, erroneous messages, i.e., messages sent in contravention of the com-

[1] The focus of [7] is on a priori assumptions that can be erroneous and may require later updates, hence, the term *a priori beliefs* there. In this paper, we generally assume these assumptions to be factive, hence, we use a priori knowledge instead.

monly known joint protocol, are usually left uninterpreted. For instance, in the epistemic modeling and analysis framework [11,16–18] for byzantine agents, message φ received from agent i is interpreted by means of the hope modality

$$H_i\varphi := correct_i \rightarrow B_i\varphi,$$

where $B_i\varphi$ represents belief of agent i and is understood in the spirit of belief as defeasible knowledge [24], where

$$B_i\varphi := K_i(correct_i \rightarrow \varphi).$$

This hope modality $H_i\varphi$ is equivalent to a disjunction

$$\neg correct_i \vee (correct_i \wedge B_i\varphi),$$

suggesting that a message φ from i is interpreted as the uncertainty between agent i being faulty or the epistemic state of i confirming φ in case i is a correct agent. Note that in the former case, the message carries no meaning whatsoever. Indeed, the axiomatization of hope in [10] takes $H_i\bot$ to be the definition of faulty agents because only a faulty agent can send contradictory messages. Given that in normal modal logic $H_i\bot \rightarrow H_i\varphi$ holds for any φ, the consequence is that a faulty agent can send any message independent of its epistemic state. In other words, no conclusions about the epistemic state of a faulty agent can be drawn from its messages, as reflected in the hope modality.

However, not all systems exhibit such a stark dichotomy between commonly known and fully transparent *us* (correct processes) and the mysterious and uninterpretable *them* (faulty processes). *Rational agents* in blockchain settings [12], for instance, do not necessarily have the same goal as the rest of the system. Nevertheless, neither their actions nor their communication are arbitrary, not to speak of adversarial. Consequently, game theoretic modeling, based on a model of their beliefs and goals, can be applied for the analysis of such systems [2].

In this paper, we extend this finer-grained view to the epistemic modeling of distributed systems and consider *heterogeneous distributed systems*, where different processes may run different protocols and where the assumption that all protocols are commonly known is dropped. In such systems, we assume that processes are partitioned into types (or roles, or classes) of agents, so that within one type the protocols are commonly known to the agents of that type. While such a strong assumption is not made for agents of different types, we do not assume them to have zero knowledge of each other's protocol either. In particular, we assume that each class is equipped with an *interpretation function* that encodes the amount of knowledge agents have regarding the preconditions for communication agents of a different type have.

Since having no preconditions for sending a message is an allowed instance, this setting generalizes the byzantine setting described earlier, where there are two types — correct and faulty agents — and only messages of correct agents have a non-trivial interpretation. These interpretation functions are formalized

by means of the new *creed* modality $\mathbb{C}_p^{A \setminus B} \varphi$ introduced in this paper, which generalizes the hope modality for the byzantine case and represents the information an agent of type A can infer upon receiving message φ from agent p of type B.

We illustrate the communication scenarios where this creed modality may be useful by means of some examples:

Example 1 ("The Murders in the Rue Morgue"). This famous story by Edgar Allan Poe describes a murder mystery. Several witnesses heard the murderer (agent m) but nobody saw m. The problem in interpreting their testimony is that they seem to contradict each other: for instance, a French witness f thinks m spoke Italian and is certain m was not French, whereas a Dutch witness d thought m was French, etc. Importantly, none of the witnesses could understand what was being said (f does not speak Italian, while d does not speak French, etc.). The standard byzantine framework considers the possibility of a faulty agent sending different messages to different agents to confuse them, but provides no means to describe one uncorrupted message being treated so differently by correct agents. Standard epistemic methods either accept all incoming information as being of equal value or make a priori preferential judgements. However, in the story, Monsieur C. Auguste Dupin correctly surmises that m spoke neither of the languages. Dupin neither dismisses witness accounts completely as lies nor accepts them completely. Instead he chooses some of the witness statements over others without prejudging them.

Example 2 (Knights and Knaves puzzles). There is a series of logical puzzles, popularized by Smullyan [26], about an island, all inhabitants of which are either *knights* who always tell the truth or *knaves* who always lie. One of the simplest ones [26, Puzzle 28] is as follows:

> There are only two people, p and q, each of whom is either a knight or a knave. p makes the following statement: "At least one of us is a knave." What are p and q?

Our goal is to incorporate the uncertainty about the mode of communication (knaves lie/knights tell the truth) into the logic. Fault-tolerant systems do not provide a satisfactory model since there information from faulty agents is either accepted (in case of benign faults) or ignored as completely unreliable (in case of byzantine faults). Instead, enough information is collected from correct agents (and they must constitute an overwhelming majority for most problems to be solvable). By contrast, knights and knaves puzzles are typically solvable even if all agents involved are knaves. The answer to the puzzle above, for instance, is that p is a knight and q is a knave. We would like to derive this answer fully within the logic.

Example 3 (Software Updates). In a highly available large scale distributed system like an ATM network, it is impossible to simultaneously update the software executed by the processes. Rather, processes are usually updated more or less sequentially during normal operation of the system, at unpredictable times. As

a consequence, the joint protocol executed in the system while a software update is in progress might mix both old and new protocol instances. Existing solutions like [1,25], which aim at updating complex protocols/software, typically provide "consistent update" environments that prevent such mixing.

Thanks to our creed modality, however, mixed joint protocols could be allowed, by explicitly considering those in the development of the new protocol instance: Indeed, when implementing a bug fix or feature update, the developer obviously knows the previous implementation. A message received at some process p from some process q in the new implementation just needs to be interpreted differently, depending on whether q runs an old or a new protocol instance. Note that backward compatibility typically rules out incorporating a version number into the messages of the (new) protocol here, in which case p would be uncertain about the actual status of q, despite having received a message from it.

For light-weight low-level protocols, this approach might indeed constitute an attractive alternative to complex consistent update mechanisms.

After introducing our framework, we explain in Sect. 6 how these examples could be formalized.

Related Work. Our logical framework generalizes the *hope* modality [10] introduced to reason about byzantine agents in distributed systems. We extend the standard formulation by considering the byzantine case as a special agent-type. Agent-types in the field of epistemic logic are formulated in [5], where *names* are used as abstract roles for groups of agents, depending on their characteristics. From the dynamic epistemic logic [9] perspective, a public announcement logic with agent types is presented in [20], providing a dynamic framework to reason about uncertainty of agent-types that is used to formalize the knights and knaves puzzle. Due to the different motivations, while treating a closely related problems set, [5] and [20] make different and at times incomparable choices regarding the postulates underlying the systems. For instance, a precondition for an announcement for an agent in [20] need not entail the agent knowing this precondition, which contradicts the fundamental Knowledge of Preconditions principle for distributed systems [22]. On the other hand, all agents in [20] possess the same knowledge about each of the existing agent types, in particular, all agents share one common interpretation of messages from a particular type, an assumption in line with the rather centralized nature of updates in dynamic epistemic logic but less sensible for distributed systems.

Paper Organization. In Sect. 2, we introduce the basic preliminary definitions and lemmas for describing heterogeneous distributed system where agents are grouped into types, each characterized by a different protocol. In Sect. 3, we provide an epistemic logic for representing heterogeneous distributed settings by introducing the *creed* modality and prove soundness and completeness in Sect. 4. We derive the properties of creed in Sect. 5. Having done that, in Sect. 6 we show how to apply this framework to the motivating examples. Finally, some conclusions are provided in Sect. 7.

2 Heterogeneous Distributed Systems

In this paper, we focus on *heterogeneous distributed systems* where agents are of different types characterized by different protocols. All agents are assumed to be at most benign faulty,[2] in the sense that they do not take actions not specified by their protocol, cannot communicate wrong information, and have perfect recall. At any time, however, agents may change their type, i.e., change their protocol.

These different protocols partition the set of processes into different types, which are identified with the names of the protocols. The set of all existing types is commonly known to all the agents. All agents of the same type, which typically work towards the same goal, use the same protocol that is commonly known to all agents of this type. What is not generally known to an agent is the distribution of agents into types and the actual protocol of a type different from its own. In other words, agent a generally does neither know the type nor the protocol of agent b.

Communication in the system is governed by the protocols. Whereas all protocols must use the same basic communication mechanism and a common layering structure [23], i.e., (possibly non-synchronous) communication rounds, agents of different types generally communicate according to different protocol rules, data formats, encodings, etc. Communication actions are triggered by preconditions that depend on the protocol of the agent's type. Consequently, the interpretation of each message depends on:

- the knowledge of the receiver about the type(s) the sender may belong to;
- the knowledge of the receiver about the communication protocol of this (these) type(s).

More formally, we consider a finite set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate with each other by using a joint communication mechanism, such as, e.g., shared memory objects or point-to-point messages. Each process executes some protocol with a name (= type) taken from a commonly known set of names $\mathcal{A}$. However, no assumption is made about the types and the actual protocols of distinct agents i and j being identical or mutually known. All protocols are organized in a common, possibly non-synchronous communication round structure. We also require that the system has a common notion of time, represented by a directed set T. Common choices for T are the set of natural number $\mathbb{N}$, or even the set of real numbers $\mathbb{R}$. It should be noted that in Definitions 4–5, we assume that concepts such as configuration and protocol match the standard notions in distributed computing literature [4,21].

Definition 4 (Heterogeneous distributed system). *We say that a tuple* $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ *is a* heterogeneous distributed system *iff*

- $\Pi = \{p_1, \ldots, p_n\}$ *is a finite set of processes;*
- $\mathcal{A} = \{A_1, \ldots, A_k\}$ *is a partition of* Π *into agent types;*

[2] Adding byzantine faults to the picture will be left for future research.

– $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ *is a collection of protocols that correspond to* $\mathcal{A}$*, one protocol per agent type;*
– $\mathcal{C}$ *is a communication medium; and*
– T *is a directed set representing global times.*

The joint protocol of $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ *is the protocol formed by the protocols of all the agents.*

In this setting, given multiple possibly non-cooperating teams of agents, we need to re-define the notion of tasks and solvability. In particular, we generally cannot impose restrictions on the output of processes in other partitions.

Definition 5 (Partial task). *We say that a tuple* $\langle S, \mathcal{I}, \mathcal{O}, \Delta \rangle$ *is a* partial task relative to $S \subseteq \Pi$ *iff* $\mathcal{I}$ *is a set of input configurations for* Π*;* $\mathcal{O}$ *is a set of output configurations for* S*; and* Δ *is a validity correspondence that maps valid initial configurations of the system to a subset of valid output configurations for* S*.*

Definition 6 (Solvability). *Let* $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ *be a heterogeneous distributed system. We say that agents of type* $A_i \in \mathcal{A}$ can solve a partial task $\mathcal{T} = \langle S, \mathcal{I}, \mathcal{O}, \Delta \rangle$ *iff for any input configuration* $\sigma \in \mathcal{I}$*, the execution of the joint protocol of* $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ *leads to an output configuration* $\rho|_S \in \Delta(\sigma)$*.*

Note that traditional distributed systems with *benign failures* fall into the particular case where $\mathcal{A} = \{\Pi\}$ and there is one unique protocol executed by all processes. Similarly, distributed systems with send-restricted byzantine faults (no false perceptions of received messages, but arbitrary message sending) could be modeled as an instance with two types $\mathcal{A}^B = \{\mathsf{Correct}, \mathsf{Faulty}\}$, where all agents of type Correct follow the intended protocol, whereas agents of type Faulty can arbitrarily deviate from it.

3 Epistemic Logic for Heterogeneous Distributed Systems

We consider a heterogeneous distributed system $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ according to Definition 4, where processes are partitioned into different types according to their protocol. Agents of the same type share a common protocol, which also includes information on how to interpret messages from agents of various types. Recall that we assume that each process knows its own protocol/type, and, therefore, the protocol of all other agents of the same type, but not necessarily which agents are of this type. In particular, an agent may be unsure whether another agent belongs to its own type or not.

Agents interpret received messages by means of an interpretation function:

Definition 7 (Interpretation function). *Let* $\mathbb{F}$ *be the set of well-defined formulas used by agents to communicate. An* interpretation function *for type* $A \in \mathcal{A}$ *with respect to type* $E \in \mathcal{A}$ *messages is any function* $f_{AE} : \mathbb{F} \to \mathbb{F}$*.*

Intuitively, $f_{AE}(\varphi)$ corresponds to the knowledge that type A agents (or simply A agents) have about the preconditions for E agents to send message φ. We assume that function f_{AE}, for every type E, is a priori known by every A agent, as part of its protocol.

Example 8. Interpretation function $f_{AE}(\varphi) := \top$ for all $\varphi \in \mathbb{F}$ corresponds to the case when A agents have no knowledge about the communication protocol of E agents. For instance, byzantine agents who can send any message at any time (send-unrestricted byzantine agents) can be captured by choosing $f_{\mathsf{Correct},\mathsf{Faulty}}(\varphi) = \top$ for partition $\mathcal{A}^E = \{\mathsf{Correct}, \mathsf{Faulty}\}$. The minimal requirement that all correct agents tell the truth translates into $f_{\mathsf{Correct},\mathsf{Correct}}(\varphi) = \varphi$.

Since we want to be able to express partition membership into our language and formulas, we need to define partition membership atoms.

Definition 9 (Propositional variables and partition atoms). *We consider, for each process $p_i \in \Pi$, a finite set $Prop_i$ of* propositional variables. *In addition, for each agent type $A \in \mathcal{A}$, we consider the set $\Pi_A := \{A_p \mid p \in \Pi\}$ of* partition atoms. *The set of all* atomic propositions *is defined as*

$$Prop := \bigcup_{i=1}^{n} Prop_i \cup \bigcup_{A \in \mathcal{A}} \Pi_A.$$

Since $\mathcal{A}$ is a partition, every agent belongs to one and only one type. For convenience, we denote the type of agent p by $\bar{p}$. Furthermore, we will assume that each agent knows its own type, i.e., $K_p(\bar{p}_p)$.

Now that we have established the basics of our heterogeneous distributed systems, we can proceed to define the language.

Definition 10 (Language of EHL). *The language $\mathcal{L}$ of the epistemic heterogeneous logic extends the standard (multi-modal) epistemic language by a new family of modalities called* creed *and is given by the grammar:*

$$\varphi ::= r \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_p\varphi, \tag{1}$$

where $r \in Prop$ is an atomic proposition (i.e., propositional variable or partition atom), $p \in \Pi$ is an agent, and $A, E \in \mathcal{A}$ are agent types. Other boolean connectives, as well as boolean constants $\top$ and $\bot$, are defined in the usual way. We use the following derived modalities: $\hat{K}_p\varphi := \neg K_p \neg\varphi$ and creed defined as

$$\mathbb{C}_p^{A \setminus E}\varphi := E_p \rightarrow K_p f_{AE}(\varphi) \tag{2}$$

for any agent $p \in \Pi$ and agent types $A, E \in \mathcal{A}$.

Creed $\mathbb{C}_p^{A \setminus E}\varphi$ represents the amount of information an A agent can extract from a message φ received from agent p under the assumption that p belongs to type E of the partition. It is based on the a priori knowledge A agents possess of the preconditions for an E agent to send message φ, as encoded in the

interpretation function f_{AE} from Definition 7, which is external to the language. This precondition already takes into account the Knowledge of Preconditions principle [22], by assuming that the sender must know that the preconditions hold. We use the standard Kripke model semantics with additional restrictions for partition atoms:

Definition 11 (Semantics). *Let $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T \rangle$ be a heterogeneous distributed system and $\{f_{AE} \mid A, E \in \mathcal{A}\}$ be the collection of interpretation functions for it. An* (epistemic) Kripke frame $F = (W, \sim)$ *is a pair of a non-empty set W of worlds (or states) and a function $\sim\colon \Pi \to \mathcal{P}(W \times W)$ that assigns to each agent $p \in \Pi$ an equivalence relation $\sim_p \subseteq W \times W$ on W. A* Kripke model $M = (W, \sim, V)$ *is a triple where $(W, \sim)$ is an epistemic Kripke frame and $V\colon W \to \mathcal{P}(Prop)$ is a* valuation function *for atomic propositions. The truth relation $\models$ between Kripke models and formulas is defined as follows: $M, s \models r$ iff $r \in V(s)$ for any $r \in Prop$; cases for the boolean connectives are standard; $M, s \models K_p\varphi$ iff $M, t \models \varphi$ for all $t \in W$ such that $s \sim_p t$. As usual, validity in a model, denoted $M \models \varphi$, means $M, s \models \varphi$ for all $s \in W$.*

A Kripke model $M = (W, \sim, V)$ is called an EHL model *iff the following two conditions hold:*

1. *For any state $s \in W$ and any agent $p \in \Pi$,*

$$\left|V(s) \cap \{A_p \mid A \in \mathcal{A}\}\right| = 1, \tag{3}$$

   i.e., exactly one of partition atoms A_p involving agent p is true at state s.
2. *For any agent p, any agent type A, and pair of states s and t,*

$$s \sim_p t \qquad \Longrightarrow \qquad \Big(A_p \in V(s) \quad \Leftrightarrow \quad A_p \in V(t)\Big), \tag{4}$$

   i.e., p can distinguish worlds where it is of different types.

General validity, denoted $\models \varphi$, means $M \models \varphi$ for all EHL *models.*

Example 12 For the interpretation functions from Example 8 for send-unrestricted byzantine agents, $\mathbb{C}_p^{\mathsf{Correct}\backslash\mathsf{Faulty}}\varphi = \mathsf{Faulty}_p \to K_p\top$. For epistemic models, it is logically equivalent to $\top$, meaning that no information can be gleaned from a message under the assumption that it is sent by a fully byzantine agent without perception flaws. At the same time, for truth-telling correct agents

$$\mathbb{C}_p^{\mathsf{Correct}\backslash\mathsf{Correct}}\varphi = \mathsf{Correct}_p \to K_p\varphi,$$

which closely matches the hope modality

$$H_p\varphi = \mathsf{Correct}_p \to K_p(\mathsf{Correct}_p \to \varphi)$$

from [10]. Indeed, since we assume agents to know their own type, it is the case that $\mathsf{Correct}_p \to K_p\mathsf{Correct}_p$ holds, making $H_p\varphi$ equivalent to $\mathbb{C}_p^{\mathsf{Correct}\backslash\mathsf{Correct}}\varphi$.

Example 13 Apart from helping to understand messages, an interpretation function can be used to gain knowledge about the type of the sender. For instance, if A agents know enough about the way E agents communicate to conclude that a particular message φ can never be sent by an E agent, which corresponds to $f_{AE}(\varphi) = \bot$, then $\mathbb{C}_q^{A\setminus E}\varphi = E_q \to K_q\bot$. For epistemic models, such $\mathbb{C}_q^{A\setminus E}\varphi$ is logically equivalent to $\neg E_q$. In other words, having received φ from agent q, an A agent p learns at least $K_p\neg E_q$.

Remark 14 (Information from message passing). Let $p, q \in \Pi$ be agents and $\mathcal{A}$ be a partition of Π. The knowledge gained by agent p upon receiving a message φ from agent q can be described by $K_p\mathbb{C}_q^p\varphi$, where

$$\mathbb{C}_q^p\varphi := \bigwedge_{E\in\mathcal{A}} \mathbb{C}_q^{\bar{p}\setminus E}\varphi \tag{5}$$

In other words, knowing its own type, p considers all possible types for the sender q and for each type considers the respective interpretation of the message; the conjunction combined with the implications within creed make sure that the appropriate type is chosen. Note that the presence of send-unrestricted agents from Example 12 adds a conjunct to (5) that is equivalent to $\top$. Hence, send-unrestricted agents can be safely ignored in determining the message meaning. By the same token, some conjuncts in (5) can rule out a particular type for agent q as in Example 13. Finally, if p has already ruled out some type E, then $K_p\neg E_q$ logically implies $K_p(E_q \to K_q f_{AE}(\varphi))$ independent of the interpretation function. In this case, the E-conjunct of (5) becomes redundant.

Example 15. In the system from Example 8 with send-unrestricted byzantine agents, upon receiving message φ from agent q, agent p can ignore the possibility of the sender being Faulty and conclude $\mathsf{Correct}_q \to K_q\varphi$, i.e., hope $H_q\varphi$ for the case of factive beliefs, in full accordance with [10]. Note also that p may infer $K_q\varphi$ from this message if p is sure that q is correct.

Now that we have established the basic definitions and semantics for the logic, we will now provide an axiomatization that we prove sound and complete in the next section.

Definition 16 (Logic EHL). *Let $\langle \Pi, \mathcal{A}, \mathcal{P}, \mathcal{C}, T\rangle$ be a heterogeneous distributed system and $\{f_{AE} \mid A, E \in \mathcal{A}\}$ be the collection of interpretation functions for it. Logic* EHL *is obtained by adding to the standard axiomatization of modal logic of knowledge* S5 *the partition axioms* P1–P3*. The resulting axiom system is as follows: for all $p \in \Pi$, all $A \in \mathcal{A}$, and all $E \in \mathcal{A}$ such that $E \neq A$,*

- Taut *All propositional tautologies in the language of* EHL*;*
- k $K_p(\varphi \to \psi) \to (K_p\varphi \to K_p\psi)$;
- 4 $K_p\varphi \to K_pK_p\varphi$;
- t $K_p\varphi \to \varphi$;
- 5 $\neg K_p\varphi \to K_p\neg K_p\varphi$;
- (MP) *rule inferring ψ from $\varphi \to \psi$ and φ;*
- (Nec) *rule inferring $K_p\varphi$ from φ;*

$$\mathsf{P1}\ \bigvee_{A\in\mathcal{A}} A_p; \qquad \mathsf{P2}\ A_p \to \neg E_p; \qquad \mathsf{P3}\ A_p \to K_p A_p. \tag{6}$$

Partition axiom P1 states that each agent belongs to at least one of the types. Partition axiom P2 postulates that each agent belongs to at most one of the types. Together they imply that agent types partition the set of agent. Partition axiom P3 expresses that every process knows its own type.

4 Soundness and Completeness of EHL

Since EHL is an extension of S5 with partition axioms governing the behavior of partition atoms while EHL models are instances of epistemic models, the soundness and completeness for EHL follows the standard proof for S5 (see, e.g., [9]), where additionally it is necessary to establish that the partition axioms are sound and that the canonical model satisfies the additional restrictions.

Theorem 17 (Soundness). *Logic* EHL *is sound with respect to* EHL *models, i.e.,* $\mathsf{EHL} \vdash \varphi$ *implies* $\models \varphi$.

Proof. We only establish the validity of partition axioms. Axioms P1 and P2 hold due to condition (3). Similarly, P3 holds because of (4). ☐

Completeness is proved by the standard canonical model construction, which requires several definitions. We omit the proofs of the following lemmas if completely standard and only treat new cases otherwise.

Definition 18 (Maximal consistent sets). *A set* $\Gamma \subseteq \mathbb{F}$ *of formulas is called* consistent *iff* $\mathsf{EHL} \nvdash \neg \bigwedge \Gamma_0$ *for any finite subset* $\Gamma_0 \subseteq \Gamma$*. A set* Γ *is called* maximal consistent *iff* Γ *is consistent but no proper superset* $\Delta \supsetneq \Gamma$ *is consistent.*

Lemma 19 (Lindenbaum Lemma). *Any consistent set* Γ *can be extended to a maximal consistent set* $\Delta \supseteq \Gamma$.

Definition 20 (Canonical model). *We define the canonical model* $M^C = (S^C, \sim^C, V^C)$ *is defined as follows:*

- S^C *is the collection of all maximal consistent sets;*
- $\Gamma \sim_p \Delta \quad$ *iff* $\quad \{K_p\varphi \mid K_p\varphi \in \Gamma\} = \{K_p\varphi \mid K_p\varphi \in \Delta\}$;
- $V^C(\Gamma) := \{r \in \mathit{Prop} \mid r \in \Gamma\}$.

Lemma 21 (Truth Lemma). *For any* $\varphi \in \mathbb{F}$ *and any* $\Gamma \in S^C$,

$$\varphi \in \Gamma \qquad \Longleftrightarrow \qquad M^C, \Gamma \models \varphi$$

Lemma 22 (Correctness). *The canonical model is an* EHL *model.*

Proof. That $S^C \neq \varnothing$ and $\sim_p$ is an equivalence relation for each $p \in \Pi$ is proved the same way as for S5. It remains to show that (3) and (4) hold.

(3) Consider any maximal consistent set $\Gamma \in S^C$ and any agent $p \in \Pi$. By the standard properties of maximal consistent sets, all theorems of EHL belong to each maximal consistent set, in particular, $\left(\bigvee_{A \in \mathcal{A}} A_p\right) \in \Gamma$ because of axiom P1. A disjunction belongs to a maximal consistent set iff one of the disjuncts does. Hence, there exists at least one type A such that $A_p \in \Gamma$. At the same time, for any other type E, we have $(A_p \to \neg E_p) \in \Gamma$ because of axiom P2. Hence, $E_p \notin \Gamma$ because maximal consistent sets are consistent and closed with respect to (MP). It follows that there is exactly one partition atom of the form A_p in Γ. Hence, by the definition of V^C,

$$\left|V^C(\Gamma) \cap \{A_p \mid A \in \mathcal{A}\}\right| = 1.$$

(4) Consider two maximal consistent sets $\Gamma \sim_p \Delta$. Let $A_p \in \Gamma$. By P3, also $K_p A_p \in \Gamma$. Hence, $K_p A_p \in \Delta$ by the definition of $\sim_p$. Finally, $A_p \in \Delta$ by axiom t. We proved that $A_p \in \Gamma$ implies $A_p \in \Delta$. The inverse implication is analogous. □

Theorem 23 (Completeness). *Logic* EHL *is complete with respect to* EHL *models, i.e.,* EHL $\vdash \varphi$ *whenever* $\models \varphi$.

Proof. We prove the contrapositive. Assume EHL $\nvdash \varphi$. That means that $\{\neg\varphi\}$ is consistent. By Lindenbaum Lemma 19, there exists a maximal consistent set $\Gamma \supseteq \{\neg\varphi\}$. Hence, this $\Gamma \in S^C$ for the canonical model M^C defined in Definition 20, which is an EHL model by Lemma 22. By the Truth Lemma 21, it follows that $M^C, \Gamma \models \neg\varphi$. Since $M^C, \Gamma \not\models \varphi$ for some EHL model, φ is not valid, i.e., $\not\models \varphi$. □

5 Properties of Creed

In this section, we derive several useful properties of creed modalities.

The explicit assumption P3 that each agent knows which type it belongs to implies a complete knowledge of own type, i.e., each agent a knows whether it belongs to any type A:

Theorem 24. EHL $\vdash \neg A_p \to K_p \neg A_p$ *for all* $p \in \Pi$, $A \in \mathcal{A}$, *i.e., agents know which type they do not belong to.*

Proof. By P1, agent p must belong to one of the types. Hence, if not type A, it must be one of the remaining types, i.e., $\neg A_p \to \bigvee_{E \neq A} E_p$. Therefore, we have $\neg A_p \to \bigvee_{E \neq A} K_p E_p$ due to P3. Given that $E_p \to \neg A_p$ for each $E \neq A$ by P2, also $K_p E_p \to K_p \neg A_p$ for each $E \neq A$ by standard modal reasoning. Hence, $\neg A_p \to K_p \neg A_p$. □

Corollary 25. EHL $\vdash K_p A_p \vee K_p \neg A_p$ *for all* $p \in \Pi$, $A \in \mathcal{A}$

Proof. It follows directly from P3 and Theorem 24 by propositional reasoning.

The creed modality amounts to K45-belief:

Theorem 26. *Creed satisfies the normality, positive and negative introspection axioms if applied to statements already translated by an interpretation function. Formally, let* $[\![\varphi]\!]_{AE}$ *stand for any formula* ξ *such that* $f_{AE}(\xi) = \varphi$*. Then the following formulas are derivable in* EHL*:*

$$\mathsf{k}_{\mathbb{C}} \vdash \mathbb{C}_p^{A\setminus E}[\![\varphi \to \psi]\!]_{AE} \to \left(\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}[\![\psi]\!]_{AE}\right)$$
$$\mathsf{4}_{\mathbb{C}} \vdash \mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}\left[\!\!\left[\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right]\!\!\right]_{AE}$$
$$\mathsf{5}_{\mathbb{C}} \vdash \neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}\left[\!\!\left[\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right]\!\!\right]_{AE}$$

Proof. We start by deriving $\mathsf{k}_{\mathbb{C}}$:

1. $\mathbb{C}_p^{A\setminus E}[\![\varphi \to \psi]\!]_{AE} = E_p \to K_p(\varphi \to \psi)$ — definition of creed
2. $K_p(\varphi \to \psi) \to (K_p\varphi \to K_p\psi)$ — axiom k
3. $\mathbb{C}_p^{A\setminus E}[\![\varphi \to \psi]\!]_{AE} \to \left(E_p \to (K_p\varphi \to K_p\psi)\right)$ — prop. reasoning from 1.,2.
4. $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} = E_p \to K_p\varphi$ — definition of creed
5. $\mathbb{C}_p^{A\setminus E}[\![\varphi \to \psi]\!]_{AE} \to \left(\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to (E_p \to K_p\psi)\right)$ — prop. reasoning from 3.,4.
6. $E_p \to K_p\psi = \mathbb{C}_p^{A\setminus E}[\![\psi]\!]_{AE}$ — definition of creed
7. $\mathbb{C}_p^{A\setminus E}[\![\varphi \to \psi]\!]_{AE} \to \left(\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}[\![\psi]\!]_{AE}\right)$ — rewriting of 5. using 6.

The following is a derivation of $\mathsf{4}_{\mathbb{C}}$:

1. $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} = E_p \to K_p\varphi$ — definition of creed
2. $K_p\varphi \to K_pK_p\varphi$ — axiom 4
3. $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to (E_p \to K_pK_p\varphi)$ — prop. reasoning from 1.,2.
4. $K_p\varphi \to (E_p \to K_p\varphi)$ — prop. tautology
5. $K_pK_p\varphi \to K_p(E_p \to K_p\varphi)$ — normal modal reasoning from 4.
6. $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \left(E_p \to K_p\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right)$ — prop. reasoning from 3.,5. using 1.
7. $E_p \to K_p\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} = \mathbb{C}_p^{A\setminus E}\left[\!\!\left[\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right]\!\!\right]_{AE}$ — definition of creed
8. $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}\left[\!\!\left[\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right]\!\!\right]_{AE}$ — rewriting of 6. using 7.

The following is a derivation of $\mathsf{5}_{\mathbb{C}}$:

1. $\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \leftrightarrow E_p \wedge \neg K_p\varphi$ — prop. reasoning from the definition of creed
2. $E_p \to K_pE_p$ — axiom P3
3. $\neg K_p\varphi \to K_p\neg K_p\varphi$ — axiom **5**
4. $\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to K_p(E_p \wedge \neg K_p\varphi)$ — normal modal reasoning from 1.–3.
5. $\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to K_p\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}$ — normal modal reasoning from 1.,4.
6. $\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \left(E_p \to K_p\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right)$ — prop. reasoning from 5.
7. $\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \mathbb{C}_p^{A\setminus E}\left[\!\!\left[\neg\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE}\right]\!\!\right]_{AE}$ — rewriting of 6. □

In addition, this creed belief is factive whenever the speaker type is correctly identified (cf. a similar conditional factivity for hope in [10]):

Theorem 27. $\mathsf{t}^*_{\mathbb{C}} : \mathsf{EHL} \vdash E_p \to \left(\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \varphi\right)$.

Proof. 1. $\vdash \mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} = (E_p \to K_p\varphi)$ definition of creed
2. $\vdash K_p\varphi \to \varphi$ axiom t
3. $\vdash E_p \to \left(\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \varphi\right)$ prop. reasoning from 1.,2. □

On the other hand, misidentifying the speaker's type may easily destroy factivity. Let $p \notin E$. Given that $\mathbb{C}_p^{A\setminus E}[\![\varphi]\!]_{AE} \to \varphi = (E_p \to K_p\varphi) \to \varphi$, we have $E_p \to K_p\varphi$ true simply because E_p is false. Accordingly, there is no reason why φ must hold.

This provides a formal model of how a true statement can lead to false beliefs due to misinterpretation. Moreover, as Theorem 26 shows, such false beliefs cannot be detected by introspection.

6 Applications

6.1 Formalizing "The Murders in the Rue Morgue"

Example 1 describes a situation where honest witnesses provide contradictory information that is, nevertheless, successfully filtered by Dupin. We show how his reasoning can be formalized and explained using the creed modality. Dupin reads all witness accounts from a paper. We assume no misinterpretation of what the witnesses said. In addition, the paper mentions the exact type of each witness (French not speaking Italian, Dutch not speaking French, etc.), which again is assumed to be factive. Hence, we use only one creed modality with the identity interpretation function per witness account read by Dupin. In other words, Dupin reasons about the available information without the need to interpret it. The crucial question is: Why does Dupin ignore some but not all of the information provided by each witness? The answer becomes clear if we view each witness account as one or several creed modalities regarding what this witness heard from m. Ignoring slight variations in details, all witness statements can be divided into two types: (a) m did not speak the language I speak; (b) m spoke a language I do not speak. Dupin accepts statements (a) but ignores statements (b). Even when statement (b) of a witness contradicts statement (a) of another witness, Dupin accepts statements (a) from both witnesses. Here is how these statements of, say, the French witness $f \in F$ regarding the utterance φ of m can be represented via the creed modality:

$$(a) \quad \mathbb{C}_m^{F\setminus F}\varphi = F_m \to K_m f_{FF}(\varphi) = F_m \to K_m\bot;$$
$$(b) \quad \mathbb{C}_m^{F\setminus I}\varphi = I_m \to K_m f_{FI}(\varphi) = I_m \to K_m\top.$$

Indeed, for (a), since the interpreting function from French to French is meaningful (in the simplest case, is the identity function), the fact that f could not understand what m was saying in this case means that $f_{FF}(\varphi) = \bot$. On the

other hand, for (b), since f does not know Italian, he has $f_{FI}(\psi) = \top$ for all ψ. As discussed in Example 13, (a) yields $\neg F_m$. Similarly, (b) yields $\top$ as per Example 12. This rightfully leads Dupin to the conclusion $\neg F_m$, i.e., $m \notin F$. In other words, statements (b) are ignored because they are trivial, not because they are false. One might say that, for f, a stronger precondition of m saying something in Italian is $m \in I$. But using $I_m \to K_m I_m$ in place of (b) would yield axiom P3, still a logically trivial statement.

In the story, m was an orangutan (Ourang-Outang in Poe's spelling), thus, fulfilling $m \notin A$ for any language A discussed.

6.2 Solution to Knights and Knaves

Clearly the partition of the island from Example 2 involves two types: I for knIghts and A for knAves. Let s be the reasoner and L be his type. The puzzle postulates that $f_{LI}(\varphi) = \varphi$ and $f_{LA}(\varphi) = \neg\varphi$ for any formula φ. Accordingly, the full information agent s receives from agent p's statement that φ is

$$\mathbb{C}_p^s \varphi = \mathbb{C}_p^{L\backslash I} \varphi \wedge \mathbb{C}_p^{L\backslash A} \varphi = (I_p \to K_p \varphi) \wedge (A_p \to K_p \neg\varphi).$$

In the puzzle in question, p states that at least one of p and q is a knave, $A_p \vee A_q$ in formulas. Hence, agent s learns

$$\mathbb{C}_p^s (A_p \vee A_q) = \Big(I_p \to K_p(A_p \vee A_q)\Big) \wedge \Big(A_p \to K_p \neg(A_p \vee A_q)\Big). \tag{7}$$

Here is how to derive in EHL that p is a knight and q is a knave, i.e., $I_p \wedge A_q$:

1. $A_p \to K_p \neg(A_p \vee A_q)$ — prop. reasoning from (7)
2. $K_p \neg(A_p \vee A_q) \to \neg A_p$ — t and prop. reasoning
3. $\neg A_p$ — prop. reasoning since $A_p \to \neg A_p$ follows from 1. and 2.
4. $\neg A_p \to I_p$ — P1 and prop. reasoning
5. I_p — (MP) from 3. and 4.
6. $I_p \to K_p(A_p \vee A_q)$ — prop. reasoning from (7)
7. $I_p \to A_p \vee A_q$ — t and prop. reasoning from 6.
8. $I_p \to A_q$ — prop. reasoning from 7. since $I_p \to \neg A_p$ by P2
9. $I_p \wedge A_q$ — prop. reasoning from 5. and 8.

Hence, $\mathsf{EHL} \vdash \mathbb{C}_p^s (A_p \vee A_q) \to I_p \wedge A_q$.

6.3 Modelling of Software Updates

Consider an heterogeneous distributed system with two agent-types, U for the updated agents running the most recent software and O for the agents running the old protocol, which is designed with the possibility of future updates in mind. Since the new protocols are designed by taking into account the existence of processes running the old protocol, the interpretation functions can be built asymmetrically. Each type interprets information from its own type directly:

$f_{UU}(\varphi) = \varphi$ and $f_{OO}(\varphi) = \varphi$. U agents can interpret messages from O agents using backward compatibility $f_{UO}(\varphi) = g\big(f_{OO}(\varphi)\big)$, where g translates into the updated system language.

The opposite is not always possible as O agents have no knowledge of the new protocols. Accordingly, messages φ compatible with the old protocol will be processed as before, i.e., using $f_{OO}(\varphi)$. But if φ is unknown to the old protocol, i.e., $f_{OO}(\varphi) = \bot$, the creed under the assumption that sender $s \in O$ would yield $\mathbb{C}_s^{O \setminus O}\varphi \leftrightarrow \neg O_s$. In this case, receiver r can conclude that the sender process s does not conform to the old protocol. Since this error flagging disappears when r is also updated, however, it may very well be the case that this does not violate the fault resilience properties of the old protocol, in particular, when not too many processes are updated simultaneously. In this case, r could be guaranteed to always compute a correct result.

6.4 Comparison to Related Work

The interpretation functions in the knights and knaves puzzles depend on the speaker only, which made it possible to formalize them in [20] by means of public announcements. In the other two examples (Rue Morgue and software update), there is an additional difficulty: even knowing the sender's type, agents interpret messages differently based on the varying levels of knowledge about the sender's protocol. This important degree of freedom of our method compared to [20] is especially central to the software update example.

7 Conclusion and Future Work

This paper provides a sound and complete axiomatization for a logic for heterogeneous distributed systems that generalizes the logic of fault-tolerant distributed systems and enables us to explicitly model the interpretation of messages sent by agents that execute different protocols (identified by types). It revolves around a (derived) new modality called creed, a generalization of the hope modality for byzantine agents, that satisfies positive and negative introspection post message-interpretation and enjoys factivity when the sender's type is correctly identified. We demonstrated the explanatory power of our approach by applying it to three representative examples from areas ranging from detective reasoning to logic puzzles to distributed systems. The current formalization assumes that agents knowledge is factive even if this factivity does not affect how they communicated. Relaxing this assumption and working with agents whose beliefs may be compromised, e.g., due to sensor errors or memory failures, is a natural next step. Another natural extension is to allow for on-the-fly updates to the interpretation functions based on received information.

Acknowledgments. The authors would like to thank Stephan Felber, Krisztina Fruzsa, Rojo Randrianomentsoa, and Thomas Schlögl as well as the participants of Dagstuhl Seminar 23272 "Epistemic and Topological Reasoning in Distributed Systems" for discussions and suggestions. We also thank the anonymous reviewers for their useful comments.

References

1. Ajmani, S., Liskov, B., Shrira, L.: Modular software upgrades for distributed systems. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 452–476. Springer, Heidelberg (2006). https://doi.org/10.1007/11785477_26
2. Amoussou-Guenou, Y., Biais, B., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Rational vs byzantine players in consensus-based blockchains. In: AAMAS 2020: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, pp. 43–51. IFAAMAS (2020). https://dl.acm.org/doi/abs/10.5555/3398761.3398772
3. Artemov, S.: Observable models. In: Artemov, S., Nerode, A. (eds.) LFCS 2020. LNCS, vol. 11972, pp. 12–26. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-36755-8_2
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. Wiley, Hoboken (2004)
5. Bílková, M., Christoff, Z., Roy, O.: Revisiting epistemic logic with names. In: Proceedings Eighteenth Conference on Theoretical Aspects of Rationality and Knowledge: Beijing, China, June 25–27, 2021. Electronic Proceedings in Theoretical Computer Science, vol. 335, pp. 39–54. Open Publishing Association (2021). https://doi.org/10.4204/eptcs.335.4
6. Castañeda, A., Gonczarowski, Y.A., Moses, Y.: Unbeatable consensus. Distrib. Comput. **35**(2), 123–143 (2022). https://doi.org/10.1007/s00446-021-00417-3
7. Cignarale, G., Schmid, U., Tahko, T., Kuznets, R.: The role of a priori belief in the design and analysis of fault-tolerant distributed systems. Mind. Mach. **33**(2), 293–319 (2023). https://doi.org/10.1007/s11023-023-09631-3
8. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design, 5th edn. Addison-Wesley, Boston (2011)
9. van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic Epistemic Logic, Synthese Library, vol. 337. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5839-4
10. Fruzsa, K., Kuznets, R., van Ditmarsch, H.: A new hope. In: Fernández-Duque, D., Palmigiano, A., Pinchinat, S. (eds.) Advances in Modal Logic, vol. 14, pp. 349–370. College Publications (2022)
11. Fruzsa, K., Kuznets, R., Schmid, U.: Fire! In: Proceedings Eighteenth Conference on Theoretical Aspects of Rationality and Knowledge: Beijing, China, June 25–27, 2021. Electronic Proceedings in Theoretical Computer Science, vol. 335, pp. 139–153. Open Publishing Association (2021). https://doi.org/10.4204/EPTCS.335.13
12. Groce, A., Katz, J., Thiruvengadam, A., Zikas, V.: Byzantine agreement with a rational adversary. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 561–572. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31585-5_50
13. Halpern, J.Y.: Using reasoning about knowledge to analyze distributed systems. Ann. Rev. Comput. Sci. **2**, 37–68 (1987). https://doi.org/10.1146/annurev.cs.02.060187.000345
14. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990). https://doi.org/10.1145/79147.79161
15. Hintikka, J.: Knowledge and Belief: An Introduction to the Logic of the Two Notions. Cornell University Press, New York (1962)
16. Kuznets, R., Prosperi, L., Schmid, U., Fruzsa, K.: Causality and epistemic reasoning in byzantine multi-agent systems. In: Moss, L.S. (ed.) Proceedings Seventeenth Conference on Theoretical Aspects of Rationality and Knowledge: Toulouse,

France, 17–19 July 2019. Electronic Proceedings in Theoretical Computer Science, vol. 297, pp. 293–312. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.297.19
17. Kuznets, R., Prosperi, L., Schmid, U., Fruzsa, K.: Epistemic reasoning with byzantine-faulty agents. In: Herzig, A., Popescu, A. (eds.) FroCoS 2019. LNCS (LNAI), vol. 11715, pp. 259–276. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29007-8_15
18. Kuznets, R., Prosperi, L., Schmid, U., Fruzsa, K., Gréaux, L.: Knowledge in byzantine message-passing systems I: framework and the causal cone. Technical report. TUW-260549, TU Wien (2019). https://publik.tuwien.ac.at/files/publik_260549.pdf
19. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982). https://doi.org/10.1145/357172.357176
20. Liu, F., Wang, Y.: Reasoning about agent types and the hardest logic puzzle ever. Mind. Mach. **23**(1), 123–161 (2013). https://doi.org/10.1007/s11023-012-9287-x
21. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, Burlington (1996)
22. Moses, Y.: Relating knowledge and coordinated action: the knowledge of preconditions principle. In: Ramanujam, R. (ed.) Proceedings Fifteenth Conference on Theoretical Aspects of Rationality and Knowledge: Carnegie Mellon University, Pittsburgh, USA, June 4–6, 2015. Electronic Proceedings in Theoretical Computer Science, vol. 215, pp. 231–245. Open Publishing Association (2015). https://doi.org/10.4204/EPTCS.215.17
23. Moses, Y., Rajsbaum, S.: A layered analysis of consensus. SIAM J. Comput. **31**(4), 989–1021 (2002). https://doi.org/10.1137/S0097539799364006
24. Moses, Y., Shoham, Y.: Belief as defeasible knowledge. Artif. Intell. **64**(2), 299–321 (1993). https://doi.org/10.1016/0004-3702(93)90107-M
25. Saur, K., Collard, J., Foster, N., Guha, A., Vanbever, L., Hicks, M.: Safe and flexible controller upgrades for SDNs. In: Symposium on Software Defined Networking (SDN) Research (SOSR 2016): March 14–15, 2016, in Santa Clara, CA. ACM (2016). https://doi.org/10.1145/2890955.2890966
26. Smullyan, R.M.: What is the Name of this Book? The Riddle of Dracula and Other Logical Puzzles. Prentice-Hall, Hoboken (1978)

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.

Symbolic Model Construction for Saturated Constrained Horn Clauses

Martin Bromberger[1], Lorenz Leutgeb[1,2](✉), and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
{mbromber,lorenz,weidenb}@mpi-inf.mpg.de

[2] Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. Clause sets saturated by hierarchic ordered resolution do not offer a model representation that can be effectively queried, in general. They only offer the guarantee of the existence of a model. We present an effective symbolic model construction for saturated constrained Horn clauses. Constraints are in linear arithmetic, the first-order part is restricted to a function-free language. The model is constructed in finite time, and non-ground clauses can be effectively evaluated with respect to the model. Furthermore, we prove that our model construction produces the least model.

Keywords: Bernays-Schönfinkel Fragment · Linear Arithmetic · Horn Clauses · Superposition · Model Construction

1 Introduction

Constrained Horn Clauses (CHCs) combine logical formulas with constraints over various domains, e.g. linear real arithmetic, linear integer arithmetic, equalities of uninterpreted functions [15]. This formalism has gained widespread attention in recent years due to its applications in a variety of fields, including program analysis and verification: safety, liveness, and termination [17,38], complexity and resource analysis [33], intermediate representation [22], and software testing [35]. Technical controls, so called *Supervisors*, like an electronic engine control unit, or a lane change assistant in a car [8,9] can be modelled, run, and proven safe. Moreover, there exist many different approaches for reasoning in CHCs and associated first-order logic fragments extended with theories [2,5,7,10,15,23–25,28,29,34,37]. Thus, CHCs are a powerful tool for reasoning about complex systems that involve logical constraints, and they have been used to solve a wide range of problems.

A failed proof attempt of some conjecture or undesired run points to a bug. In this case investigation of the cause of the unexpected result or behavior is crucial. Building a model of the situation that can then be effectively queried is an important means towards a repair. However, some algorithms for CHCs,

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 137–155, 2023.
https://doi.org/10.1007/978-3-031-43369-6_8

e.g. hierarchic superposition, which boils down to hierarchic ordered resolution in the context of CHCs, do not return a model that can be effectively queried if a proof attempt fails, in general. If so, queries are still restricted to ground clauses [4].

The contribution of our paper can be seen as an extension for these saturation based algorithms that produces models and not just saturated clause sets. In fact, we show how to build symbolic models out of any saturated CHC clause set over linear arithmetic. This fragment is equivalent to Horn clause sets of linear arithmetic combined with the Bernays-Schönfinkel fragment. Recall that although satisfiability in this fragment is undecidable [16,26], in general, for a finitely saturated set we can construct such a representation in finite time.

Our models fulfill all important properties postulated in the literature for automated model building in first-order logic [13,20]. First, they can be *effectively constructed*, i.e., each model is represented by one linear arithmetic formula of finite size for each of its predicates and it can be constructed in finite time. Second, they are *unique*, i.e., the model representation specifies exactly one interpretation; in our case the least model. Third, they can be *effectively queried*, i.e., we provide decision procedures that evaluate whether an atom, clause, or formula is entailed/satisfied by the model. Fourth, it is possible to *test* the *equivalence* of two models. The approach we present does not exploit features of linear arithmetic beyond equality, the existence of a well-founded order for the theories' universe, and decidability of the theory. The results may therefore be adapted to other constraint domains. Model representation that can be effectively constructed and queried like ours are also called *effective model representations*. Moreover, our method is the first effective model construction approach for ordered resolution (or its extension to superposition) that is based on saturation, goes beyond ground clauses, and includes theory constraints. In the future, we plan to use this approach as the basis for a more general model construction approach that also works on more expressive fragments of first-order logic modulo theories.

Our model construction is inspired by the model construction operator used in the proof for refutational completeness of hierarchic superposition [3,6,30]. The main difference is that the model construction operator from the refutational completeness proof is restricted to ground clauses and executed on the potentially infinite ground instances of the saturated clause set (in addition to an infinite axiomatization of the background theory as ground clauses). As a result, the model construction operator from the refutational completeness proof cannot effectively construct a model because iterating over a potentially infinite set means it may diverge. Moreover, in contrast to our model construction, the original model operator cannot effectively evaluate non-ground atoms, clauses, or formulas. It is, however, sufficient, to show the existence of a model if the clause set is saturated and does not contain the empty clause [3,6,30]. In our version of the model construction operator, we managed to lift the restriction to ground clause sets by restricting the input logic to the Horn Bernays-Schönfinkel fragment instead of full first-order logic. This enables us to define a strict propagation/production order for our non-ground clauses instead of just for ground clauses. As a result, we can construct the model one clause at a time.

The paper is organized as follows. In Sect. 2 we clarify notation and preliminaries. The main contribution is presented in Sect. 3. At the end of this section, we also explain how our models satisfy the postulates (see [13, Section 5.1, p. 234]) by Fermüller and Leitsch for automated model building. We conclude in Sect. 4. Proofs were elided in favor of explanations and examples. An extended version, which includes proofs, can be found at [12].

2 Preliminaries and Notation

We briefly recall the basic logical formalisms and notations we build upon [9]. Our starting point is a standard first-order language with *variables* (denoted x, y, z), *predicates* (denoted P, Q) of some fixed *arity*, and *terms* (denoted t, s). An *atom* (denoted A) is an expression $P(t_1, \dots, t_n)$ for a predicate P of arity $n = \text{arity}(P)$. When the terms $t_1, \dots, t_n$ in $P(t_1, \dots, t_n)$ are not relevant in some context, we also write $P(*)$. A *positive literal* is an atom A and a *negative literal* is a negated atom $\neg A$. We define $\text{comp}(A) = \neg A$, $\text{comp}(\neg A) = A$, $|A| = A$ and $|\neg A| = A$. Literals are usually denoted L, K. We sometimes write literals as $[\neg]P(*)$, meaning that the sign of the literal is arbitrary, often followed by a case distinction. Formulas are defined in the usual way using quantifiers $\forall$, $\exists$ and the boolean connectives (in order of decreasing binding strength) $\neg$, $\vee$, $\wedge$, $\rightarrow$, and $\leftrightarrow$. The logic we consider does not feature a first-order equality predicate.

A *clause* (denoted C, D) is a universally closed disjunction of literals $A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$. We may equivalently write $B_1 \wedge \dots \wedge B_m \rightarrow A_1 \vee \dots \vee A_n$. A clause is *Horn* if it contains at most one positive literal, i.e. $n \leq 1$. In Sect. 3, all clauses considered are Horn clauses. If Y is a term, formula, or a set thereof, $\text{vars}(Y)$ denotes the set of all variables in Y, and Y is *ground* if $\text{vars}(Y) = \emptyset$. Analogously, $\Pi(Y)$ is the set of predicate symbols occurring in Y.

The *Bernays-Schönfinkel Clause Fragment* (BS) in first-order logic consists of first-order clauses where all terms are either variables or constants. The *Horn Bernays-Schönfinkel Clause Fragment* (HBS) is further restricted to Horn clauses.

A *substitution* σ is a function from variables to terms with a finite domain and codomain. We denote substitutions by σ, τ. The application of substitutions is often written postfix, as in $x\sigma$, and is homomorphically extended to terms, atoms, literals, clauses, and quantifier-free formulas. A substitution is *ground* if its codomain is ground. Let Y denote some term, literal, clause, or clause set. A substitution σ is a *grounding* for Y if $Y\sigma$ is ground, and $Y\sigma$ is a *ground instance* of Y in this case. We denote by $\text{gnd}(Y)$ the set of all ground instances of Y. The *most general unifier* $\text{mgu}(Z_1, Z_2)$ of two terms/atoms/literals Z_1 and Z_2 is defined as usual, and we assume that it does not introduce fresh variables and is idempotent.

2.1 Horn Bernays-Schönfinkel with Linear Arithmetic

The class HBS(LRA) is the extension of the Horn Bernays-Schönfinkel fragment with linear real arithmetic (LRA). Analogously, the classes HBS(LQA) and

HBS(LIA) are the extensions of the Horn Bernays-Schönfinkel fragment with linear rational arithmetic (LQA) and linear integer arithmetic (LIA), respectively. The only difference between the three classes are the sort LA their variables and terms range over and the universe $\mathcal{U}$ over which their interpretations range. As the names already imply LA = LRA and $\mathcal{U} = \mathbb{R}$ for HBS(LRA), LA = LQA and $\mathcal{U} = \mathbb{Q}$ for HBS(LQA), and LA = LIA and $\mathcal{U} = \mathbb{Z}$ for HBS(LIA). The results presented in this paper hold for all three classes and by HBS(LA) we denote that we are talking about an arbitrary one of them.

Linear arithmetic terms are constructed from a set $\mathcal{X}$ of *variables*, the set of constants $c \in \mathbb{Q}$ (if in HBS(LRA) or HBS(LQA)) or $c \in \mathbb{Z}$ (if in HBS(LIA)), and binary function symbols $+$ and $-$ (written infix). Additionally, we allow multiplication $\cdot$ if one of the factors is a constant. Multiplication only serves us as syntactic sugar to abbreviate other arithmetic terms, e.g., $x + x + x$ is abbreviated to $3 \cdot x$. Atoms in HBS(LA) are either *first-order atoms* (e.g., $P(13, x)$) or *(linear) arithmetic atoms* (e.g., $x < 42$). Arithmetic atoms are denoted by λ and may use the predicates $\leq, <, \approx, \not\approx, >, \geq$, which are written infix and have the expected fixed interpretation. We use $\approx$ instead of $=$ to avoid confusion between equality in LA and equality on the meta level. While we do not permit quantifiers in the syntax of clauses, the notion of symbolic interpretations that we will develop does require this, denoted as usual. By atoms(Y)/quants(Y) we denote the linear arithmetic atoms/quantifiers in a formula or set of formulas Y. *First-order literals* and related notation is defined as before. *Arithmetic literals* coincide with arithmetic atoms, since the arithmetic predicates are closed under negation, e.g., $\neg(x \geq 42)$ is equivalent to $x < 42$.

HBS(LA) clauses are defined as for HBS but using HBS(LA) atoms. We often write clauses in the form $\Lambda \parallel C$ where C is a clause solely built of free first-order literals and Λ is a multiset of LA atoms called the *constraint* of the clause. A clause of the form $\Lambda \parallel C$ is therefore also called a *constrained clause*. Since the interpretation of linear arithmetic relations is fixed, we set $\Pi(\Lambda \parallel C) \coloneqq \Pi(C)$.

The fragment we consider in Sect. 3 is restricted even further to *abstracted* clauses: For any clause $\Lambda \parallel C$, all terms in C must be variables. Put differently, we disallow any arithmetic function symbols, including numerical constants, in C. Variable abstraction, e.g. rewriting $x \geq 3 \parallel P(x, 1)$ to $x \geq 3, y \approx 1 \parallel P(x, y)$, is always possible. Hence, the restriction to abstracted clauses is not a theoretical limitation, but allows us to formulate our model construction operator in a more concise way. We assume abstracted clauses for theory development, but we prefer non-abstracted clauses in examples for readability, e.g., a unit clause $P(3, 5)$ is considered in the development of the theory as the clause $x \approx 3, y \approx 5 \parallel P(x, y)$.

In contrast to other works, e.g. [11], we do not permit first-order constants, and consequently also no variables that range over the induced Herbrand universe. All variables are arithmetic in the sense that they are interpreted by $\mathcal{U}$. Since we only allow equalities in the arithmetic constraint, it is possible to simulate variables over first-order constants, by e.g. numbering them, i.e. defining a bijection between $\mathbb{N}$ and constant symbols. So this again not a theoretical limitation.

The semantics of $\Lambda \,\|\, C$ is as follows:

$$\Lambda \,\|\, C \quad \text{iff} \quad \big(\bigwedge_{\lambda \in \Lambda} \lambda\big) \rightarrow C \quad \text{iff} \quad \big(\bigvee_{\lambda \in \Lambda} \neg\lambda\big) \vee C$$

For example, the clause $x > 1 \vee y \not\approx 5 \vee \neg Q(x) \vee R(x,y)$ is also written $x \leq 1, y \approx 5 \,\|\, \neg Q(x) \vee R(x,y)$. The negation $\neg(\Lambda \,\|\, C)$ of a constrained clause $\Lambda \,\|\, C$ where $C = A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_m$ is thus equivalent to $(\bigwedge_{\lambda \in \Lambda} \lambda) \wedge \neg A_1 \wedge \cdots \wedge \neg A_n \wedge B_1 \wedge \cdots \wedge B_m$. Note that since the neutral element of conjunction is $\top$, an empty constraint is thus valid, i.e. equivalent to true. In analogy to the empty clause in settings without constraints, we write $\square$ to mean any and all clauses $\Lambda \,\|\, \bot$ where Λ is satisfiable, which are all unsatisfiable.

An *assignment* for a constraint Λ is a substitution (denoted β) that maps all variables in $\text{vars}(\Lambda)$ to values in $\mathcal{U}$. An assignment is a *solution* for a constraint Λ if all atoms $\lambda \in (\Lambda\beta)$ evaluate to true. A constraint Λ is *satisfiable* if there exists a solution for Λ. Otherwise it is *unsatisfiable*.

We assume *pure* input clause sets because otherwise satisfiability is undecidable for impure HBS(LA) [21]. This means the only constants of our sort LA are concrete rational numbers. Irrational numbers are not allowed by the standard definition of the theory. Fractions are not allowed if LA = LIA. Satisfiability of pure HBS(LA) clause sets is semi-decidable, e.g., using *hierarchic superposition* [3] or *SCL(T)* [10]. Note that pure HBS(LA) clauses correspond to *constrained Horn clauses (CHCs)* with LA as background theory.

All arithmetic predicates and functions are interpreted in the usual way denoted by the interpretation $\mathcal{A}^{\text{LA}}$. An interpretation of HBS(LA) coincides with $\mathcal{A}^{\text{LA}}$ on arithmetic predicates and functions, and freely interprets non-arithmetic predicates. For pure clause sets this is well-defined [3]. Logical satisfaction and entailment is defined as usual, and uses similar notation as for HBS.

Example 1. The clause $y \geq 5, x' \approx x + 1 \,\|\, S_0(x,y) \rightarrow S_1(x',0)$ is part of a timed automaton with two clocks x and y modeled in HBS(LA). It represents a transition from state S_0 to state S_1 that can be traversed only if clock y is at least 5 and that resets y to 0 and increases x by 1.

2.2 Ordering Literals and Clauses

In order to define redundancy for constrained clauses, we need an *order*: Let $\prec_\Pi$ be a total, well-founded, strict ordering on predicate symbols and let $\prec_\mathcal{U}$ be a total, well-founded, strict ordering on the universe $\mathcal{U}$. (Note that $\prec$ cannot be the standard ordering $<$ because it is not well-founded for $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. In the case of $\mathbb{R}$, the existence of such an order is even dependent on whether we assume the axiom of choice [18].) We extend these orders step by step. First, to atoms, i.e., $P(\vec{a}) \prec Q(\vec{b})$ if $P \prec_\Pi Q$ or $P = Q$, $\vec{a}, \vec{b} \in \mathcal{U}^{|\vec{a}|}$, and $\vec{a} \prec_{\text{lex}} \vec{b}$, where $\prec_{\text{lex}}$ is the lexicographic extension of $\prec_\mathcal{U}$. Next, we extend the order to literals with a strict precedence on the predicate and the polarity, i.e.,

$$P(\vec{t}) \prec \neg P(\vec{s}) \prec Q(\vec{u}) \qquad \text{if } P \prec Q$$

independent of the arguments of the literals. Then, take the multiset extension to order clauses. To handle constrained clauses extend the relation such that constraint literals (in our case arithmetic literals) are always smaller than first-order literals. We conflate the notation of all extensions into the symbol $\prec$ and define $\preceq$ as the reflexive closure of $\prec$. Note that $\prec$ is only total for ground atoms/literals/clauses, which is sufficient for a hierarchic superposition order [6].

Definition 2 ($\prec$-maximal Literal). *A literal L is called* $\prec$-maximal *in a clause C if there exists a grounding substitution σ for C, such that there is no different $L' \in C$ for which $L\sigma \prec L'\sigma$. The literal L is called* strictly $\prec$-maximal *if there is no different $L' \in C$ for which $L\sigma \preceq L'\sigma$.*

Proposition 3. *If $\prec$ is a predicate-based ordering, C is a Horn clause, C has a positive literal L, and L is $\prec$-maximal in C, then L is strictly $\prec$-maximal in C.*

Definition 4 ($\prec$-maximal Predicate in Clause). *A predicate symbol P is called* (strictly) $\prec$-maximal *in a clause C if there is a literal $[\neg]P(*) \in C$ that is (strictly) $\prec$-maximal in C.*

Definition 5. *Let N be a set of clauses, $\prec$ a clause ordering, C a clause, and P a predicate symbol. Then $N^{\prec C} := \{C' \in N \mid C' \prec C\}$ and $N^{\preceq P} := \{C \in N \mid Q$ is $\prec$ -maximal in C and $Q \preceq P\}$.*

2.3 Hierarchic Superposition, Redundancy and Saturation

For pure HBS(LA) most rules of the (hierarchic) superposition calculus become obsolete or can be simplified. In fact, in the HBS(LA) case (hierarchic) superposition boils down to (hierarchic) ordered resolution. For a full definition of (hierarchic) superposition calculus in the context of linear arithmetic, consider SUP(LA) [1]. Here, we will only define its simplified version in the form of the hierarchic resolution rule.

Definition 6 (Hierarchic $\prec$-Resolution). *Let $\prec$ be an order on literals and $\Lambda_1 \parallel L_1 \vee C_1$, $\Lambda_2 \parallel L_2 \vee C_2$ be constrained clauses. The inference rule of hierarchic $\prec$-resolution is:*

$$\frac{\Lambda_1 \parallel L_1 \vee C_1 \qquad \Lambda_2 \parallel L_2 \vee C_2 \qquad \sigma = \mathrm{mgu}(L_1, \mathrm{comp}(L_2))}{(\Lambda_1, \Lambda_2 \parallel C_1 \vee C_2)\sigma}$$

where L_1 is $\prec$-maximal in C_1 and L_2 is $\prec$-maximal in C_2.

Note that in the resolution rule we do not enforce explicitly that the positive literal is strictly maximal. This is possible because in the Horn case any positive literal is strictly maximal if it is maximal in the clause.

For saturation, we need a termination condition that defines when the calculus under consideration cannot make any further progress. In the case of superposition, this notion is that any new inferences are *redundant*.

Definition 7 (Clause Redundancy). *A ground clause $\Lambda \parallel C \in N$ is* redundant *with respect to a set N of ground clauses and order $\prec$ if $N^{\prec \Lambda \parallel C} \models \Lambda \parallel C$. A potentially non-ground clause $\Lambda \parallel C \in N$ is* redundant *with respect to a potentially non-ground clause set N and order $\prec$ if for all $\Lambda' \parallel C' \in \operatorname{gnd}(\Lambda \parallel C)$ the clause $\Lambda' \parallel C'$ is redundant with respect to* $\operatorname{gnd}(N)$.

If a clause $\Lambda \parallel C \in N$ is redundant with respect to a clause set N, then it can be removed from N without changing its semantics. If $\Lambda \parallel C$ is newly inferred, then we also call it redundant if $\Lambda \parallel C$ is already part of N. The same cannot be said for clauses in N or all clauses in N would be redundant. Determining clause redundancy is an undecidable problem [10,40]. However, there are special cases of redundant clauses that can be easily checked, e.g., tautologies and subsumed clauses. Redundancy also means that $\mathcal{I} \models N^{\prec \Lambda \parallel C}$ implies $\mathcal{I} \models \Lambda \parallel C$ if $\Lambda \parallel C$ is redundant w.r.t. N. We will exploit this fact in the model construction.

Definition 8 (Saturation). *A set of clauses N is* saturated up to redundancy *with respect to some set of inference rules, if application of any rules to clauses in N yields a clause that is redundant with respect to N or is contained in N.*

2.4 Interpretations

In our context, models are interpretations that satisfy (sets of) clauses. The standard notion of an interpretation is fairly opaque and interprets a predicate P as the potentially infinite set of ground arguments that satisfy P.

Definition 9 (Interpretation). *Let P be a predicate symbol with* $\operatorname{arity}(P) = n$. *Then, $P^{\mathcal{I}}$ denotes the subset of $\mathcal{U}^n$ for which the* interpretation *$\mathcal{I}$ maps the predicate symbol P to* true.

Since our model construction approach manipulates interpretations directly, we need a notion of interpretations that always has a finite representation and for which it is possible to decide (in finite time) whether a clause is satisfied by the interpretation. Therefore, we rely on the notion of symbolic interpretations:

Definition 10 (Symbolic Interpretation). *Let $x_1, x_2, \ldots$ be an infinite sequence of distinct variables, i.e. $x_i \neq x_j$ for all $1 \leq i < j$. (We assume the same sequence for all symbolic interpretations in order to prevent conflicts when we later combine multiple symbolic interpretations into one.) A* symbolic interpretation *$\mathcal{S}$ is a function that maps every predicate symbol P with* $\operatorname{arity}(P) = n$ *to a formula denoted $P^{\mathcal{S}}(\vec{x})$ of finite size, constructed using the usual boolean connectives over* LA *atoms, where the only free variables appear in $\vec{x} = (x_1, \ldots, x_n)$. The interpretation $\mathcal{I}_{\mathcal{S}}$ corresponding to $\mathcal{S}$ is defined by $P^{\mathcal{I}_{\mathcal{S}}} = \{(\vec{x})\beta \mid \beta \models P^{\mathcal{S}}(\vec{x})\}$ and maps the predicate symbol P to* true *for the subset of $\mathcal{U}^n$ which corresponds to the solutions of $P^{\mathcal{S}}(\vec{x})$.*

Example 11. Let N be a clause set consisting of the clauses $0 \leq x \leq 2, 0 \leq y \leq 2 \| P(x, y)$ and $x_Q \geq x_P + 1, y_Q \geq y_P + 1 \| \neg P(x_P, y_P) \vee Q(x_Q, y_Q)$. An example

of a symbolic interpretation $\mathcal{S}$ that satisfies N, would be the function that maps P to $P^{\mathcal{S}}(x_1, x_2) = 0 \leq x_1 \leq 2 \wedge 0 \leq x_2 \leq 2$ and $Q^{\mathcal{S}}(x_1, x_2) = 1 \leq x_1 \wedge 1 \leq x_2$. It corresponds to the interpretation $\mathcal{I}_{\mathcal{S}}$ where $P^{\mathcal{I}_{\mathcal{S}}} = \{(a_1, a_2) \in \mathcal{U} \mid 0 \leq a_1 \leq 2 \wedge 0 \leq a_2 \leq 2\}$ and $Q^{\mathcal{I}_{\mathcal{S}}} = \{(a_1, a_2) \in \mathcal{U} \mid 1 \leq a_1 \wedge 1 \leq a_2\}$.

The notion of symbolic interpretations is closely related to *$\mathcal{A}$-definable models* [7, Definition 7] and *constrained atomic representations* [13, Definition 5.1, pp. 236–237]. Each symbolic interpretation $\mathcal{S}(\vec{x})$ is equivalent to a constrained atomic representation that consists of one constraint atom $[[P(\vec{x}) : P^{\mathcal{S}}(\vec{x})]]$ (written in the notation from [13]) for every predicate P. Note that in this context the constraint is not just a quantifier-free conjunction of linear arithmetic atoms, but a linear arithmetic formula potentially containing quantifiers (although those can be eliminated with quantifier elimination techniques).

Due to the fact that each symbolic interpretation consists of a finite set of formulas of finite size, symbolic interpretations can be considered as finite representations. In contrast, the standard representation of an interpretation as a potentially infinite set of ground atoms is not a finite representation. However, this also means that there are some interpretations for which no corresponding symbolic interpretation exists, for instance the set of prime numbers is a satisfying interpretation for $y \approx 2 \,\|\, P(y)$, but not expressible as a symbolic interpretation (in LA). As we will later see, at least any saturated set of HBS(LA) clauses either is unsatisfiable or has a symbolic interpretation that satisfies it (Theorem 29).

The *top interpretation*, denoted $\mathcal{I}_\top$, is defined as $P^{\mathcal{I}_\top} := \mathcal{U}^n$ for all predicate symbols P with $\text{arity}(P) = n$ and corresponds to the *top symbolic interpretation*, denoted $\mathcal{S}_\top$, defined as $P^{\mathcal{S}_\top} := \top$ for all predicate symbols P. The *bottom interpretation* (or *empty interpretation*), denoted $\mathcal{I}_\bot$, and the *bottom symbolic interpretation* (or *empty symbolic interpretation*), denoted $\mathcal{S}_\bot$, are defined analogously. The interpretation of P under $\mathcal{I} \cup \mathcal{J}$ is defined as $P^{\mathcal{I} \cup \mathcal{J}} := P^{\mathcal{I}} \cup P^{\mathcal{J}}$ for every predicate P. In the symbolic case, $\mathcal{S} \cup \mathcal{R}$ is defined as $P^{\mathcal{S} \cup \mathcal{R}}(\vec{x}) := P^{\mathcal{S}}(\vec{x}) \vee P^{\mathcal{R}}(\vec{x})$ for every predicate P. We write $\mathcal{I} \subseteq \mathcal{J}$ or $\mathcal{I}$ is *included in* $\mathcal{J}$ (resp. $\mathcal{I} \subset \mathcal{J}$ or $\mathcal{I}$ is *strictly* included in $\mathcal{J}$) if $P^{\mathcal{I}} \subseteq P^{\mathcal{J}}$ (resp. $P^{\mathcal{I}} \subset P^{\mathcal{J}}$) for all predicate symbols P.

Definition 12 (Entailment of Literal). *Let $\mathcal{I}$ be an interpretation. Given a ground literal $P(a_1, \ldots, a_n)$, where $a_i \in \mathcal{U}$, we write $\mathcal{I} \models P(a_1, \ldots, a_n)$ if $(a_1, \ldots, a_n) \in P^{\mathcal{I}}$. Conversely, we write $\mathcal{I} \not\models P(a_1, \ldots, a_n)$ if $(a_1, \ldots, a_n) \notin P^{\mathcal{I}}$. For a non-ground literal L, we write $\mathcal{I} \models L$ if for all grounding substitutions σ for L, we have $\mathcal{I} \models L\sigma$. Conversely, we write $\mathcal{I} \not\models L$, if there exists a grounding substitution σ for L, such that $\mathcal{I} \not\models L\sigma$.*

We overload $\models$ for symbolic interpretations, i.e. we write $\mathcal{S} \models L$ and mean $\mathcal{I}_{\mathcal{S}} \models L$. The following function encodes a clause as an LA formula for evaluation under a given symbolic interpretation.

Definition 13 (Clause Evaluation Function). *Let $\Lambda \,\|\, C$ be a constrained clause where $C = L_1 \vee \cdots \vee L_m$, $L_i = [\neg]P_i(y_{i,1}, \ldots, y_{i,n_i})$ and let $\mathcal{S}$ be a symbolic interpretation. Then the clause evaluation function $(\Lambda \,\|\, C)^{\mathcal{S}}$ is defined as*

follows based on the definitions for σ_i and ϕ_i (for $1 \leq i \leq m$):

$$\sigma_i := \{x_j \mapsto y_{i,j} \mid 1 \leq j \leq n_i\} \qquad \phi_i := \begin{cases} P_i^{\mathcal{S}} & L_i \text{ is positive} \\ \neg P_i^{\mathcal{S}} & L_i \text{ is negative (otherwise)} \end{cases}$$

$$\big(\Lambda \,\|\, C\big)^{\mathcal{S}} := \big(\bigwedge_{\lambda \in \Lambda} \lambda\big) \to \big(\bigvee_{i=1}^{m} \phi_i \sigma_i\big)$$

Note that the free variables of $(\Lambda \,\|\, C)^{\mathcal{S}}$ are exactly the free variables of $(\Lambda \,\|\, C)$. Moreover, the substitutions σ_i are necessary in the above definition in order to map the variables in the symbolic interpretation for the predicates $P_i^{\mathcal{S}}$ to the variables that appear as arguments in the literals $P_i(y_{1,1}, \ldots, y_{1,n_i})$.

Proposition 14. *Given a constrained clause $\Lambda \,\|\, C$ with grounding β, we have*

$$\vDash \big(\Lambda \,\|\, C\big)^{\mathcal{S}} \beta \qquad \text{if and only if} \qquad \mathcal{S} \vDash \big(\Lambda \,\|\, C\big) \beta$$

As a corollary of the previous proposition, the entailment $\mathcal{S} \vDash \Lambda \,\|\, C$ holds if and only if the universal closure of the formula $(\Lambda \,\|\, C)^{\mathcal{S}}$ is valid. This means that for a symbolic interpretation $\mathcal{S}$ it is always computable whether a clause is entailed by $\mathcal{S}$ because there are decision procedures for quantified LRA, LQA, and LIA formulas of finite size.

We require two functions that manipulate LA-formulas directly to express our model construction (cf. Definition 17), i.e. to map solutions for a clause defined by a formula vars(ϕ) to one atom inside the clause. This requires from us to project away all variables in ϕ that appear in the clause but not in the atom.

Definition 15 (Projection). *Let V be a set of variables and ϕ be an* LA-*formula. The projection function π is defined as follows:*

$$\pi(V, \phi) := \exists x_1 \ldots \exists x_n.\, \phi \quad \text{where } \{x_1, \ldots, x_n\} = \mathrm{vars}(\phi) \setminus V$$

$\pi(V, \phi)$ is a standard projection function that binds a subset V of the variables in the formula ϕ with existential quantifiers. Note that we also know that $\pi(V, \phi)$ is equivalent to a quantifier-free LA formula just over the variables $x_1, \ldots, x_n$ because there exist quantifier elimination algorithms for LRA, LQA, and LIA [14,32].

A further function $\curlyvee$ is needed when we encounter literals of the form $P(x, x, \ldots)$, i.e., where one variable is shared among two arguments. In this case, we use $\curlyvee$ to express in our symbolic interpretation that the equivalent argument positions must also be equivalent in our interpretation.

Definition 16 (Sharing). *Let $(y_1, \ldots, y_n)$ and $(x_1, \ldots, x_n)$ be tuples of variables with the same length. The sharing function $\curlyvee$, which encodes variable sharing across different argument positions, is defined as follows:*

$$\curlyvee\big((y_1, \ldots, y_n), (x_1, \ldots, x_n)\big) := \bigwedge_{1 \leq i < j \leq n,\; y_i = y_j} x_i \approx x_j$$

2.5 Consequence and Least Model

The notion of a *least model* is common in logic programming. Horn logic programs admit a least model, which is the intersection of all models of the program (see [31, § 6, p. 36]). In our context, the least model of a set of clauses N is the intersection of all models of N. An alternative characterization of the least model of N is through the least fixed point of the one-step consequence operator, which we define as T_N for the context of LA constraints analogously to [27, Section 4]. The one-step consequence operator T_N takes a set of clauses N and an interpretation $\mathcal{I}$ as input and returns an interpretation:

$$P^{T_N(\mathcal{I})} := \left\{ (\vec{y})\beta \;\middle|\; \begin{array}{l} \Lambda \,\|\, \neg P_1(\vec{y_1}) \vee \cdots \vee \neg P_n(\vec{y_n}) \vee P(\vec{y}) \in N, \\ \models \Lambda\beta, \text{and } \mathcal{I} \models P_i(\vec{y_i})\beta \text{ for } 1 \leq i \leq n \end{array} \right\}$$

The least fixed point of this operator exists by Tarski's Fixed Point Theorem [39]: Interpretations form a complete lattice under inclusion (supremum given by union, infimum given by intersection), and T_N is monotone.

3 Model Construction

In this section we address construction of models for HBS(LA). Throughout this section, we consider a set of constrained Horn clauses N and an order $\prec$ to be given. Our aim is to define an interpretation $\mathcal{I}_N$, such that

$$\mathcal{I}_N \models N \qquad \text{if } N \text{ is saturated and } \square \notin N$$

Towards that goal, we define the operator $\delta(\mathcal{S}, \Lambda \,\|\, C' \vee P(\vec{y}))$. It takes a symbolic interpretation $\mathcal{S}$, and a Horn clause with maximal literal $P(\vec{y})$. It results in a symbolic interpretation that accounts for $\Lambda \,\|\, C' \vee P(\vec{y})$.

Definition 17 (Production Operator). *Let $\Lambda \,\|\, C$ be a constrained Horn clause, where $C = C' \vee P(\vec{y})$, $P(\vec{y}) \succ C'$, and $C' = \neg P_1(y_{1,1}, \ldots, y_{1,n_1}) \vee \cdots \vee \neg P_m(y_{m,1}, \ldots, y_{m,n_m})$. Let $\mathcal{S}$ be a symbolic interpretation, where the free variables of $P^{\mathcal{S}}$ are $\vec{x}$ and the free variables of $P_i^{\mathcal{S}}$ are $\vec{x_i}$ (for $1 \leq i \leq m$). Note that $n = |\vec{y}| = |\vec{x}| = \text{arity}(P)$.*

The production operator $\delta(\mathcal{S}, \Lambda \,\|\, C)$ *results in a new symbolic interpretation*

$$\begin{aligned} P^{\delta(\mathcal{S},\Lambda \,\|\, C)}(\vec{x}) &:= \Big(\pi\big(\{y_1, \ldots, y_n\}, \bigwedge_{\lambda \in \Lambda} \lambda \wedge \bigwedge_{i=1}^{m} (P_i^{\mathcal{S}})\sigma_i\big)\Big)\sigma \wedge \curlyvee\big(\vec{y}, \vec{x}\big) \\ Q^{\delta(\mathcal{S},\Lambda \,\|\, C)}(\vec{z}) &:= \bot \qquad \text{for all } Q \neq P \text{ where } |\vec{z}| = \text{arity}(Q) \end{aligned}$$

where, to map variables from literal arguments to the variables appearing in the symbolic interpretation $\mathcal{S}$ and back, we have the substitutions

$$\begin{aligned} \sigma &:= \{y' \mapsto x_j \mid y' \in \{y_1, \ldots, y_n\} \text{ and } j \text{ is the smallest index s.t. } y_j = y'\} \\ \sigma_i &:= \{x_{i,j} \mapsto y_{i,j} \mid 1 \leq j \leq n_i\} \qquad \text{for } 1 \leq i \leq m \end{aligned}$$

The goal of the operator $\delta(\mathcal{S}, \Lambda \parallel C)$ is to define an extension of the symbolic interpretation $\mathcal{S}$ such that $\mathcal{S} \cup \delta(\mathcal{S}, \Lambda \parallel C)$ satisfies $\Lambda \parallel C$. Note that δ only extends the interpretation over the strictly maximal predicate P. Moreover, due to our predicate order, it only needs to consider the interpretation $\mathcal{S}$ for predicates Q with $Q \prec P$. δ also satisfies the following two symmetrical properties: On the one hand, every grounding τ of $\Lambda \parallel C' \vee P(\vec{y})$ that is not yet satisfied by $\mathcal{S}$ must correspond to solution β of $P^{\delta(\mathcal{S}, \Lambda \parallel C' \vee P(\vec{y}))}$ that satisfies $P(\vec{y})\tau$. On the other hand, every solution β of $P^{\delta(\mathcal{S}, \Lambda \parallel C' \vee P(\vec{y}))}$ must correspond to a grounding of $\Lambda \parallel C' \vee P(\vec{y})$ that is not yet satisfied by $\mathcal{S}$. The first property is needed so $\mathcal{S} \cup \delta(\mathcal{S}, \Lambda \parallel C' \vee P(\vec{y}))$ satisfies $\Lambda \parallel C' \vee P(\vec{y})$. The second property is needed so we do not accidentally extend our interpretation by any solutions not needed to satisfy $\Lambda \parallel C' \vee P(\vec{y})$.

Note that in the above statements β and τ are generally not the same because the variables $\vec{x}$ used to define $P^{\mathcal{S}}$ are not necessarily the same as the variables appearing in the clause $\Lambda \parallel C$ and literal $P(\vec{y})$. There are three reasons for this that are handled by three different methods in our model construction:

1. The variables in $\mathcal{S}$ and $\Lambda \parallel C$ simply do not match, e.g. in $P^{\mathcal{S}} := x_1 \approx 0$ and $\Lambda \parallel C := y_1 > 0 \parallel P(y_1)$. This is handled by the substitution σ in δ that maps all variables in $P(\vec{y})$ to their appropriate variables in $P^{\mathcal{S}}$, e.g. in the previous example $\sigma = \{y_1 \mapsto x_1\}$ and $P^{\delta(\mathcal{S}, \Lambda \parallel C)} = (y_1 > 0)\sigma = x_1 > 0$.
2. Not all variables in $\Lambda \parallel C$ also appear in $P(\vec{y})$, e.g. in $P^{\mathcal{S}} := x_1 \approx 0$ and $\Lambda \parallel C := x_1 \approx y_1 + 1 \wedge y_1 \approx 0 \parallel P(x_1)$. This is handled in δ by the projection operator π (Definition 15) that binds all variables that appear in $\Lambda \parallel C$ but not in $P(\vec{y})$, e.g. in the previous example $P^{\delta(\mathcal{S}, \Lambda \parallel C)} := \pi(\{y_1\}, x_1 \approx y_1 + 1 \wedge y_1 \approx 0)$, where $\pi(\{y_1\}, x_1 \approx y_1 + 1 \wedge y_1 \approx 0) = \exists y_1.\ x_1 \approx y_1 + 1 \wedge y_1 \approx 0$, which is equivalent to $x_1 \approx 1$.
3. Some variables might occur in multiple argument positions, e.g. in $\Lambda \parallel C := \top \parallel P(y_1, y_1)$. This case is covered in δ by the sharing function $\curlyvee$ (c.f. Definition 16) that expresses which variables in $P^{\delta(\mathcal{S}, \Lambda \parallel C)}$ must map to the same value. Continuing the example, $\curlyvee((y_1, y_1), (x_1, x_2)) = x_1 \approx x_2$ and $P^{\delta(\mathcal{S}, \Lambda \parallel C)}(x_1, x_2) = \curlyvee((y_1, y_1), (x_1, x_2))$.

The parts of $P^{\delta(\mathcal{S}, \Lambda \parallel C)}$ that we have not yet discussed are based on the fact that any constrained Horn clause $\Lambda \parallel C' \vee P(\vec{y})$ can also be written as an implication of the form $\phi \rightarrow P(\vec{y})$, where $\phi := \Lambda \wedge P_1(y_{1,1}, \ldots, y_{1,n_1}) \wedge \cdots \wedge P_m(y_{m,1}, \ldots, y_{m,n_m})$ and $\mathcal{S} \not\models \Lambda \parallel C'\tau$ if and only if $\mathcal{S} \models \phi\tau$. This means the groundings τ of $\Lambda \parallel C'$ not satisfied by $\mathcal{S}$ are also the groundings of ϕ satisfied by $\mathcal{S}$. It is straightforward to express these groundings with a conjunctive formula based on Λ and the $P_i^{\mathcal{S}}$. The only challenge is the reverse problem from before, i.e. mapping the variables of $P_i^{\mathcal{S}}$ to the variables in the literals $P_i(y_{1,1}, \ldots, y_{1,n_i})$. This mapping is done in δ by the substitution σ_i.

Now, based on the production operator δ for one clause, we can use an inductive definition over the order $\prec$ to define an interpretation $\mathcal{S}_N$ for all clauses in N. We distinguish the following auxiliary symbolic interpretations: $\mathcal{S}_{\prec P}$ which captures progress up to but excluding the predicate P, Δ_P which captures how P should be interpreted considering $\mathcal{S}_{\prec P}$, and $\mathcal{S}_{\preceq P}$ which captures progress

up to and including the predicate P. The symbolic interpretation $\Delta_P^{\Lambda \parallel C}$ is the extension of $\mathcal{S}_{\prec P}$ w.r.t. the single clause $\Lambda \parallel C$.

Definition 18 (Model Construction). *Let N be a finite set of constrained Horn clauses. We define symbolic interpretations $\mathcal{S}_{\prec P}$, $\mathcal{S}_{\preceq P}$ and Δ_P for all predicates $P \in \Pi(N)$ by mutual induction over $\prec$:*

$$\mathcal{S}_{\preceq P} := \mathcal{S}_{\prec P} \cup \Delta_P \qquad \mathcal{S}_{\prec P} := \bigcup_{Q \prec P} \Delta_Q \qquad \Delta_P := \bigcup_{\Lambda \parallel C' \vee P(*) \in N} \Delta_P^{\Lambda \parallel C' \vee P(*)}$$

$$\Delta_P^{\Lambda \parallel C} := \begin{cases} \delta(\mathcal{S}_{\prec P}, \Lambda \parallel C) & \text{if } P(\vec{y}) \text{ maximal in } C, \text{ and } \mathcal{S}_{\prec P} \nvDash \Lambda \parallel C \\ \mathcal{S}_\perp & \text{otherwise} \end{cases}$$

Finally, based on the above inductive definition of $\mathcal{S}_{\prec P}$ for every predicate symbol $P \in \Pi(N)$, we arrive at an overall interpretation for N.

Definition 19 (Candidate Interpretation). *The* candidate interpretation *for N (w.r.t $\prec$), denoted $\mathcal{I}_N$, is the interpretation associated with the symbolic interpretation $\mathcal{S}_N = \bigcup_{P \in \Pi(N)} \Delta_P$ where P ranges over all predicate symbols occurring in N.*

Note that $\mathcal{S}_N = \mathcal{S}_{\preceq P}$ where P is $\prec$-maximal in $\Pi(N)$. Obviously, we intend that $\mathcal{S}_N \vDash N$ if N is saturated (Theorem 29). Otherwise, i.e. $\mathcal{S}_N \nvDash N$, we can use our construction to find a non-redundant inference (Corollary 30). Consider the following two examples, demonstrating how δ sits at the core of the aforementioned inductive definitions of symbolic interpretations.

Example 20 (Dependent Interpretation). Assume $P \prec Q$ and consider the following set of clauses:

$$N := \left\{ \begin{array}{lll} 0 \le y_1 \le 2, 0 \le y_2 \le 2 & \parallel \underline{P(y_1, y_2)} & (C_1), \\ y_3 \ge y_1 + 1, y_4 \ge y_2 + 1 & \parallel P(y_1, y_2) \rightarrow \underline{Q(y_3, y_4)} & (C_2) \end{array} \right\}$$

Maximal literals are underlined. Since the maximal literals of C_1 and C_2 are both positive, ordered resolution cannot be applied. The set is saturated. Since P is the $\prec$-smallest predicate we have $\mathcal{S}_{\prec P} = \mathcal{S}_\perp$. Applying the δ operator yields the following interpretation for P:

$$P^{\mathcal{S}_{\preceq P}} = P^{\delta(\mathcal{S}_{\prec P}, C_1)}(x_1, x_2) = 0 \le x_1 \le 2 \wedge 0 \le x_2 \le 2$$

Then, Q is interpreted relative to P. Consider the clause C_2: For all solutions of its constraint $y_3 \ge y_1 + 1, y_4 \ge y_2 + 1$ our model must also satisfy its logical part $P(y_1, y_2) \rightarrow Q(y_3, y_4)$. The intuition that Q depends on P arises from the implication in the logical part. Whenever the constraint of C_2 and $P(y_1, y_2)$ are satisfied, $Q(y_3, y_4)$ must be satisfied. These are exactly the points defined through $\delta(\mathcal{S}_{\prec Q}, C_2)$, based on $\mathcal{S}_{\prec Q} = \mathcal{S}_{\preceq P} = \delta(\mathcal{S}_{\prec P}, C_1)$:

$$\begin{aligned} Q^{\delta(\mathcal{S}_{\prec Q}, C_2)}(x_1, x_2) &= \exists z_1, z_2.\ x_1 \ge z_1 + 1 \wedge x_2 \ge z_2 + 1 \wedge 0 \le z_1 \le 2 \wedge 0 \le z_2 \le 2 \\ &= x_1 \ge 1 \wedge x_2 \ge 1 \end{aligned}$$

Whenever the conjuncts $0 \leq y_1 \leq 2$ and $0 \leq y_2 \leq 2$ are satisfied, the premise of the implication is true, thus there must be a solution to the interpretation of Q, additionally abiding the constraint of the clause. Since Q is $\prec$-maximal in N, we arrive at $\mathcal{S}_N = \mathcal{S}_{\preceq Q} = \mathcal{S}_{\preceq P} \cup \delta(\mathcal{S}_{\prec Q}, C_2) = \delta(\mathcal{S}_\perp, C_1) \cup \delta(\mathcal{S}_{\preceq P}, C_2)$. See Fig. 1a for a visual representation of $\mathcal{S}_N$.

Example 21 (Unsaturated Clause Set). Assume $P \prec Q$ and consider the following set of clauses:

$$N := \left\{ \begin{array}{llll} y_1 < 0 \,\|\, \underline{P(y_1)} & (C_1), & y_1 < 1 \,\|\, \underline{Q(y_1)} & (C_3), \\ y_1 > 0 \,\|\, \underline{P(y_1)} & (C_2), & y_1 \leq 0 \,\|\, \underline{Q(y_1)} \rightarrow P(y_1) & (C_4) \end{array} \right\}$$

Maximal literals are underlined. Note that a resolution inference is possible, since the maximal literals of C_3 and C_4 have opposite polarity, use the same predicate symbol, and are trivially unifiable. Thus, in this example we consider the effect of applying our model construction to a clause set that is *not* saturated. Since P is $\prec$-minimal, we start with the following steps:

$$\begin{array}{lll} \mathcal{S}_{\prec P} = \mathcal{S}_\perp & P^{\delta(\mathcal{S}_{\prec P}, C_1)}(x_1) = x_1 < 0 & \\ & P^{\delta(\mathcal{S}_{\prec P}, C_2)}(x_1) = x_1 > 0 & P^{\mathcal{S}_{\preceq P}}(x_1) = x_1 < 0 \vee x_1 > 0 \end{array}$$

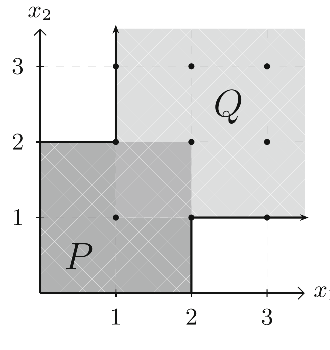
Next, we obtain the following results for Q:

$$\begin{array}{lll} \mathcal{S}_{\prec Q} = \mathcal{S}_{\preceq P} & Q^{\delta(\mathcal{S}_{\prec Q}, C_3)}(x_1) = x_1 < 1 & \\ & Q^{\delta(\mathcal{S}_{\prec Q}, C_4)}(x_1) = \perp & Q^{\mathcal{S}_{\preceq Q}}(x_1) = x_1 < 1 \vee \perp = x_1 < 1 \end{array}$$

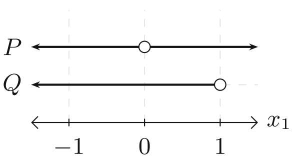
See Fig. 1b for a visual representation of $\mathcal{S}_N = \mathcal{S}_{\preceq Q}$. Note that $\mathcal{S}_N \nvDash C_4$, since we have $\mathcal{S}_N \vDash Q(0)$ but $\mathcal{S}_N \nvDash P(0)$. Thus, by using the constructed model, we can pinpoint clauses that contradict that N is saturated. Applying resolution to C_3 and C_4 leads to the clause $y_1 \leq 0 \,\|\, P(y_1)$ labelled C_5. If we then add C_5 to N, we instead get $P^{\mathcal{S}_{\preceq P}}(x_1) = x_1 < 0 \vee x_1 > 0 \vee x_1 \leq 0 = \top$.

In the following, we clarify some properties of the construction. We provide an upper bound for the number of LA atoms and quantifiers in the symbolic model for LRA and LQA. Although we do not state it explicitly, the estimate for LIA works in a similar way, but due to the higher complexity of LIA quantifier elimination, the size of the symbolic model grows triple exponentially [36].

Proposition 22. *If N is a finite set of* LRA/LQA *constrained Horn clauses, and $\mathcal{S}'_N$ the result of applying quantifier elimination to $\mathcal{S}_N$ then, for every predicate symbol $P \in \Pi(N)$, the number of* LA *atoms in $P^{\mathcal{S}'_N}$ is in $O(m^{2 \cdot q^{p-1}} \cdot n^{2 \cdot q^{p-1}} \cdot (l + a^2)^{q^p})$ where n is the max. number of clauses with the same max. predicate, m is the max. number of non-arithmetic literals in a clause, l is the max. number of arithmetic literals in a clause, a is the max. arity of any predicate, $p = |\Pi(N)|$, q is the max. difference of variables in any clause and its positive maximal literal.*



(a) Result of Example 20.



(b) Result of Example 21.

Fig. 1. Visual representation of the models resulting from Examples 20 and 21.

Corollary 23 (Effective Construction). *If N is a finite set of constrained Horn clauses then for every predicate $P \in \Pi(N)$, $P^{\mathcal{S}_N}$ is a linear arithmetic formula of finite size, and can be computed in a finite number of steps.*

We show that all points in $P_N^{\mathcal{I}}$ are necessary and justified in some sense, that $\mathcal{I}_N$ is indeed a model of N, and that $\mathcal{I}_N$ is also the least model of N if N is saturated. The notion of whether a clause is productive captures whether it contributes something to the symbolic interpretation.

Definition 24 (Productive Clause). *Let P be a predicate symbol with* arity$(P) = n$*. We say that $\Lambda \,\|\, C$* produces *$P(a_1, \ldots, a_n)$ if $(a_1, \ldots, a_n) \in P^{\Delta_P^{\Lambda \,\|\, C}}$.*

Next, we want to formally express that every element of the resulting interpretation is justified. Firstly, we express that the operator δ will produce points such that every clause is satisfied whenever necessary, i.e. whenever the maximal literal of the clause is $P(*)$ and the maximal literal not satisfied by $\mathcal{S}_{\prec P}$.

Proposition 25. *Let $\Lambda_C \,\|\, C$ where $C = C' \vee P(\vec{y})$ and $C' \prec P(\vec{y})$. Let τ be a grounding substitution for $\Lambda_C \,\|\, C$. If $\mathcal{S}_{\prec P} \not\models (\Lambda_C \,\|\, C)\tau$, then $\models \Lambda_C\tau$ and $\mathcal{S}_{\preceq P} \models P(\vec{y})\tau$, thus $\mathcal{S}_{\preceq P} \models (\Lambda_C \,\|\, C)\tau$.*

Secondly, we express that for every point in $P_N^{\mathcal{I}}$, it is justified in the sense that there is a clause that produced the point, i.e. this clause would otherwise not be satisfied by the resulting interpretation.

Proposition 26. *If $\mathcal{S}_{\preceq P} \models P(\vec{a})$, then there exists a clause $\Lambda_C \,\|\, C$ where $C = C' \vee P(\vec{y})$ and $C' \prec P(\vec{y})$, and there exists a grounding τ for $\Lambda_C \,\|\, C$, such that $P(\vec{a}) = P(\vec{y})\tau$ and $\mathcal{S}_{\prec P} \not\models (\Lambda_C \,\|\, C)\tau$.*

Also, observe that once the maximal predicate P of a given clause is interpreted by $\mathcal{S}_{\preceq P}$, the interpretation of the clause does not change for $\mathcal{S}_{\preceq Q}$ where $Q \succ P$.

Corollary 27. *Let $P \prec Q \preceq R$, and P be maximal in clause C. If $\mathcal{S}_{\preceq P} \models \Lambda_C \,\|\, C$ or $\mathcal{S}_{\prec Q} \models \Lambda_C \,\|\, C$, then $\mathcal{S}_{\prec R} \models \Lambda_C \,\|\, C$ and $\mathcal{S}_{\preceq R} \models \Lambda_C \,\|\, C$.*

As a result, we know that the full model satisfies N, i.e., $\mathcal{I}_N \models N$ if every clause is satisfied at the point of the construction, where the interpretation of its maximal predicate P stays fixed.

Proposition 28. *For every clause $\Lambda_C \,\|\, C \in N$ with maximal predicate P, if $\mathcal{S}_{\preceq P} \models \Lambda_C \,\|\, C$, then $\mathcal{I}_N \models N$.*

With the above propositions (and some auxiliary properties that can be found in [12]) we show that indeed $\mathcal{I}_N \models N$ if N is saturated and does not contain the empty clause.

Theorem 29. *Let $\prec$ be a clause ordering and N be a set of constrained Horn clauses. If (1.) N is saturated w.r.t. $\prec$-resolution, and (2.) $\square \notin N$, then $\mathcal{I}_N \models N$.*

For clauses with positive maximal literal, the fact that they are satisfied by $\mathcal{I}_N$ follows from Proposition 25. For clauses with maximal literal $\neg P(*)$, we prove this theorem by contradiction: If there is a minimal clause $\Lambda_C \,\|\, C$ such that $\mathcal{S}_N \not\models \Lambda_C \,\|\, C$. We can then exploit Proposition 26 to find the smallest clause $\Lambda_D \,\|\, D$ that produced the respective instance $P(\vec{a})$. Applying hierarchic $\prec$-resolution to $\Lambda_C \,\|\, C$ and $\Lambda_D \,\|\, D$ then yields a non-redundant clause. This idea then leads to the following theorem.

Corollary 30. *Let $\prec$ be a clause ordering and N be a set of constrained Horn clauses. If (1.) $\mathcal{I}_N \not\models N$, and (2.)$\square \notin N$, then there exist two clauses $\Lambda_C \,\|\, C$, $\Lambda_D \,\|\, D \in N$ such that: (1.) $\Lambda_C \,\|\, C$ is the smallest clause not satisfied by $\mathcal{I}_N$, i.e. there exists a grounding τ such that $\mathcal{I}_N \not\models (\Lambda_C \,\|\, C)\tau$, but there does not exist a clause $\Lambda_{C'} \,\|\, C' \in N$ with grounding τ', such that $\mathcal{I}_N \not\models (\Lambda_{C'} \,\|\, C')\tau'$ and $(\Lambda_{C'} \,\|\, C')\tau' \prec (\Lambda_C \,\|\, C)\tau$, (2.)$\neg P(\vec{a})$ is the maximal literal of $(\Lambda_C \,\|\, C)\tau$, (3.)$\Lambda_D \,\|\, D$ is the minimal clause that produces $P(\vec{a})$, (4.)$\prec$-resolution is applicable to $\Lambda_C \,\|\, C$ and $\Lambda_D \,\|\, D$, and (5.)the resolvent of $\Lambda_C \,\|\, C$ and $\Lambda_D \,\|\, D$ is not redundant w.r.t. N.*

Additionally, we show that $\mathcal{I}_N$ is the least model of N, establishing a connection between our approach and the literature on constrained Horn clauses (see [27, Section 4] and [15, Section 2.4.1]) and logic programming (see [31, § 6, p. 37]).

Theorem 31. *$\mathcal{I}_N$ is the least model of N.*

Fermüller and Leitsch define four postulates (see [19] as cited in [13, Section 5.1, p. 234]) regarding *automated model building*. In the following, we instantiate the postulates for our setting. By $\mathfrak{S}(N)$ we denote the set of all symbolic interpretations of the set of constrained Horn clauses N. We argue how our approach satisfies all postulates, one by one:

Uniqueness. *Each element of $\mathfrak{S}(N)$ specifies a single interpretation of N.*
We have shown (cf. Theorem 31) that $\mathcal{I}_N$, the model represented by $\mathcal{S}_N$, is the least model of N, which is unique.

Atom Test. *There exists a fast procedure to evaluate arbitrary ground atoms over $\Pi(N)$ in the interpretation defined by a $\mathcal{S}$ in $\mathfrak{S}(N)$.*
This is a special case of clause evaluation (cf. Proposition 14): A ground atom $P(\vec{t})$ is true in $\mathcal{S}$ if and only if $\models P^{\mathcal{S}}(\vec{x})\{x_i \mapsto t_i \mid 1 \leq i \leq |\vec{x}| = |\vec{t}|\}$. Fulfillment of this property thus hinges on the meaning of "fast". We consider methods for evaluating formulas of LA against points to be fast.

Formula Evaluation. *There exists an algorithm deciding the truth values of arbitrary formulas in interpretations defined by $\mathcal{S} \in \mathfrak{S}(N)$.*
Proposition 14 states that evaluating a constrained clause $\Lambda \parallel C$ is achieved by evaluating the universal closure of $(\Lambda \parallel C)^{\mathcal{S}}$, which is decided by quantifier elimination algorithms for LRA, LQA, and LIA [14,32]. For sets of clauses, evaluate each clause individually and combine the results conjunctively.

Equivalence Test. *There exists an algorithm which decides whether two representations $\mathcal{S}_1$ and $\mathcal{S}_2$ in $\mathfrak{S}(N)$ describe the same interpretation.*
$\mathcal{S}_1$ and $\mathcal{S}_2$ describe the same interpretation if and only if for each predicate $P \in \Pi(N)$ of arity n, we have $\forall x_1 \ldots \forall x_n. P^{\mathcal{S}_1}(\vec{x}) \leftrightarrow P^{\mathcal{S}_2}(\vec{x})$.

4 Conclusion

We have presented the first model construction approach to Horn clauses with linear arithmetic constraints based on hierarchic ordered resolution, (cf. Definition 19). The linear arithmetic constraints may range over the reals, rationals, or integers. The computed model is the canonical least model of the saturated Horn clause set (cf. Theorem 31). Clauses can be effectively evaluated with respect to the model (cf. Proposition 14). This offers a way to explore the properties of a saturated clause set, e.g., if the set represents a failed refutation attempt.

Future Work. It is straightforward to see that any symbolic LQA model is also a symbolic LRA model. (This holds due to convexity of conjunctions of ground LQA atoms.) So even if the axiom of choice is not assumed, there is an alternative way to obtain a model for a HBS(LRA) clause set: Simply treat it as an HBS(LQA) clause set, saturate it and construct its model based on HBS(LQA).

In this work, we restrict ourselves to only one sort LA per set of clauses. An extension to a many-sorted setup, e.g. including first-order variables with sort $\mathcal{F}$ is possible. This can even be simulated, by encoding first-order constants as concrete natural numbers via a bijection to $\mathbb{N}$, since $\mathbb{N} \subset \mathcal{U}$. By not placing any arithmetic constraints on the variables used for the encoding, it can be read off and mapped back from the resulting model.

One obvious challenge is relaxation of the restriction to Horn clauses. With respect to ordered resolution saturation there is typically no difference in the sense that if a Horn fragment can always be finitely saturated, so can the non-Horn fragment be. However, our proposed ordering for the model construction at the granularity of predicate symbols will not suffice in this general case, and the key to overcome this challenge seems to be the appropriate treatment of clauses

with maximal literals of the same predicate. Backtracking on the selection of literals might also be sufficient.

The approach we presented does not exploit features of linear arithmetic beyond equality and the existence of a well-founded order for the underlying universe $\mathcal{U}$. The results may therefore be adapted to other constraint domains such as non-linear arithmetic.

Acknowledgements. We thank our reviewers for their constructive comments.

References

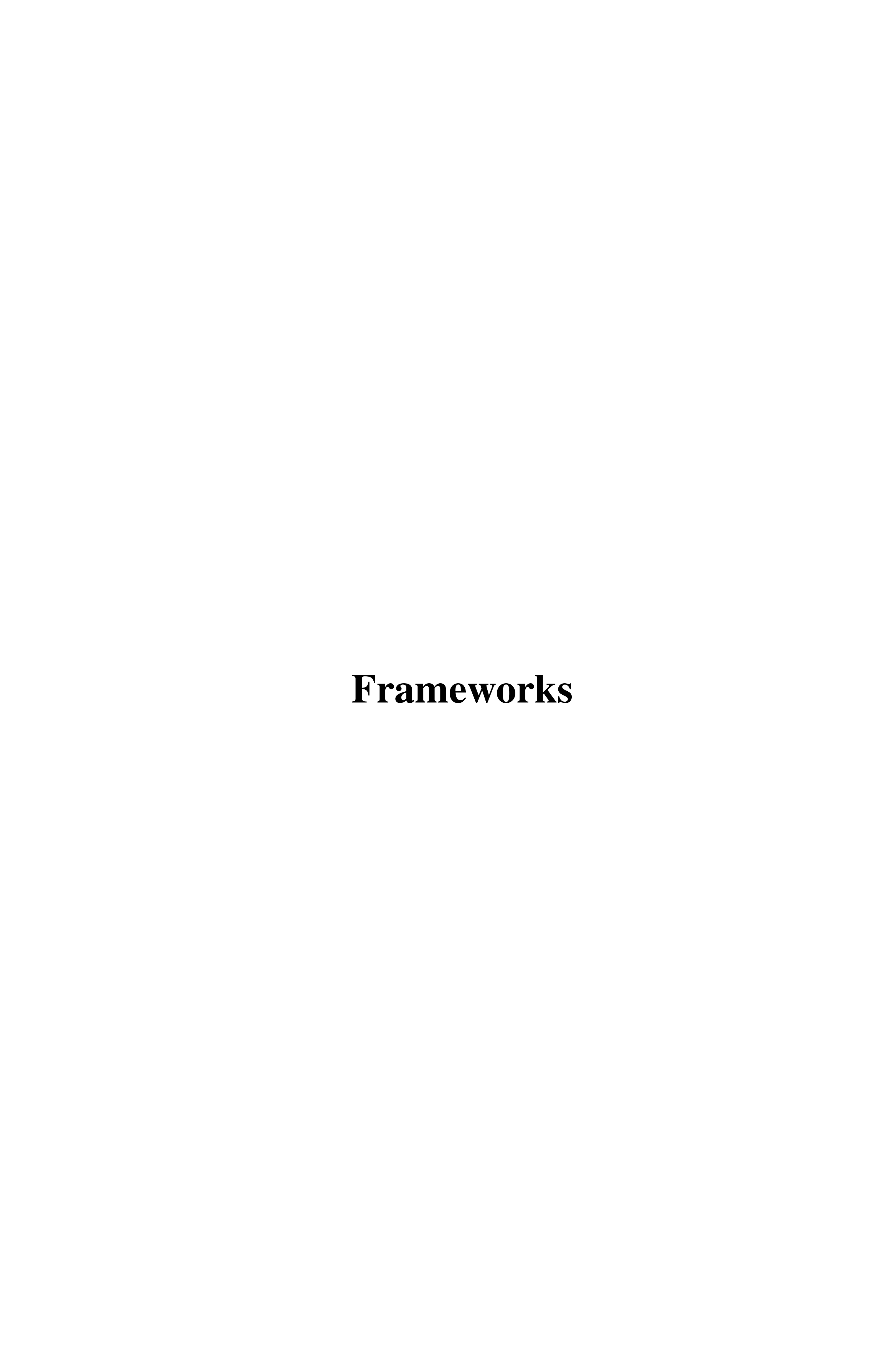
1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 84–99. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_5
2. Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition 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. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. AAECC **5**, 193–212 (1994). https://doi.org/10.1007/BF01190829
4. Basin, D.A., Ganzinger, H.: Automated complexity analysis based on ordered resolution. JACM **48**(1), 70–109 (2001). https://doi.org/10.1145/363647.363681
5. Baumgartner, P., Fuchs, A., Tinelli, C.: (LIA) - model evolution with linear integer arithmetic constraints. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 258–273. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_19
6. Baumgartner, P., Waldmann, U.: Hierarchic superposition revisited. In: Lutz, C., Sattler, U., Tinelli, C., Turhan, A.-Y., Wolter, F. (eds.) Description Logic, Theory Combination, and All That. LNCS, vol. 11560, pp. 15–56. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22102-7_2
7. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
8. Bromberger, M., et al.: A sorted datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: TACAS 2022. LNCS, vol. 13243, pp. 480–501. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_27
9. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., Krötzsch, M., Weidenbach, C.: A datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: Konev, B., Reger, G. (eds.) FroCoS 2021. LNCS (LNAI), vol. 12941, pp. 3–24. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86205-3_1

10. 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
11. Bromberger, M., Leutgeb, L., Weidenbach, C.: An efficient subsumption test pipeline for BS(LRA) clauses. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 147–168. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_10
12. Bromberger, M., Leutgeb, L., Weidenbach, C.: Symbolic model construction for saturated constrained horn clauses. arXiv (2023). https://doi.org/10.48550/arXiv.2305.05064
13. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building, APLS, vol. 31. Springer, Dordrecht (2004). https://doi.org/10.1007/978-1-4020-2653-9
14. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Mach. Intell. **7**, 91–99 (1972)
15. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained horn clauses for program verification. TPLP **22**(6), 974–1042 (2022). https://doi.org/10.1017/S1471068421000211
16. Downey, P.J.: Undecidability of presburger arithmetic with a single monadic predicate letter. Center for Research in Computer Technology, Harvard University, Technical report (1972)
17. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-guided termination analysis. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 124–143. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_7
18. Feferman, S.: Some applications of the notions of forcing and generic sets. Fundamenta Mathematicae. **56**(3), 325–345 (1964). http://eudml.org/doc/213821
19. Fermüller, C.G., Leitsch, A.: Hyperresolution and automated model building. LOGCOM **6**(2), 173–203 (1996). https://doi.org/10.1093/logcom/6.2.173
20. Fermüller, C.G., Leitsch, A.: Decision procedures and model building in equational clause logic. IGPL **6**(1), 17–41 (1998). https://doi.org/10.1093/jigpal/6.1.17
21. Fiori, A., Weidenbach, C.: SCL with theory constraints. arXiv (2020). http://arxiv.org/abs/2003.04627
22. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Horn clauses as an intermediate representation for program analysis and transformation. TPLP **15**(4–5), 526–542 (2015). https://doi.org/10.1017/S1471068415000204
23. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: 14th LICS, 1999, pp. 295–303. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782624
24. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416. ACM (2012). https://doi.org/10.1145/2254064.2254112
25. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
26. Horbach, M., Voigt, M., Weidenbach, C.: The universal fragment of presburger arithmetic with unary uninterpreted predicates is undecidable. arXiv (2017). http://arxiv.org/abs/1703.01212
27. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. JLP **19**(20), 503–581 (1994). https://doi.org/10.1016/0743-1066(94)90033-7

28. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
29. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74915-8_19
30. Kruglov, E.: Superposition modulo theory. Ph.D. thesis, Saarland University (2013). http://scidok.sulb.uni-saarland.de/volltexte/2013/5559/
31. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Cham (1987). https://doi.org/10.1007/978-3-642-83189-8
32. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. **36**(5), 450–462 (1993). https://doi.org/10.1093/comjnl/36.5.450
33. López-García, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. TPLP **18**(2), 167–223 (2018). https://doi.org/10.1017/S1471068418000042
34. McMillan, K.L.: Lazy annotation revisited. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 243–259. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_16
35. Mesnard, F., Payet, É., Vidal, G.: Concolic testing in CLP. TPLP **20**(5), 671–686 (2020). https://doi.org/10.1017/S1471068420000216
36. Oppen, D.C.: A 2ˆ2ˆ2ˆPN upper bound on the complexity of Presburger arithmetic. JCSS **16**(3), 323–332 (1978). https://doi.org/10.1016/0022-0000(78)90021-1
37. Rümmer, 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
38. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. TOPLAS **32**(3), 8:1-8:70 (2010). https://doi.org/10.1145/1709093.1709095
39. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**(2), 285–309 (1955). https://doi.org/10.2140/pjm.1955.5.285
40. Weidenbach, C.: Automated reasoning building blocks. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 172–188. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_12

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.



Frameworks

Combining Finite Combination Properties: Finite Models and Busy Beavers

Guilherme V. Toledo[1(✉)], Yoni Zohar[1], and Clark Barrett[2]

[1] Bar-Ilan University, Ramat Gan, Israel
guivtoledo@gmail.com
[2] Stanford University, Stanford, USA

Abstract. This work is a part of an ongoing effort to understand the relationships between properties used in theory combination. We here focus on including two properties that are related to shiny theories: the finite model property and stable finiteness. For any combination of properties, we consider the question of whether there exists a theory that exhibits it. When there is, we provide an example with the simplest possible signature. One particular class of interest includes theories with the finite model property that are not finitely witnessable. To construct such theories, we utilize the Busy Beaver function.

Keywords: satisfiability modulo theories · theory combination · theory politeness · theory shininess

1 Introduction

The story of this paper begins with [7], where it was shown that the theory of algebraic datatypes, useful for modeling data structures like lists and trees, can be combined with any other theory, using the polite combination method [6]. This combination method offers a way to combine decisions procedures of two theories into a decision procedure for the combined theory, with different assumptions than those of the earlier Nelson-Oppen approach [4]. In particular, it was proven that the theory admits a technical property concerning cardinalities of models, called *strong politeness* [2]. It was noted in [7] that proving strong politeness for this theory seemed much harder than proving *politeness*, a similar but simpler property. Therefore, the proof was split into three steps: (i) a class of theories was identified in which politeness and strong politeness coincide; (ii) the theory of algebraic datatypes was shown to be in this class; and (iii) this theory was proven to be polite. This proof technique raised the following question: **does politeness imply strong politeness?** An affirmative answer to this question would simplify strong politeness proofs that follow such steps, as only the last step would be needed. Unfortunately, the answer to this question was shown in [8] to be negative, in its most general form. However, an affirmative answer was given for theories over one-sorted empty signatures, where politeness and strong politeness do coincide.

Seeing that relationships between model-theoretic properties of theories (like politeness and strong politeness) are non-trivial, and can have a big impact on proofs in the

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 159–175, 2023.
https://doi.org/10.1007/978-3-031-43369-6_9

field of theory combination, we have recently initiated a more general research plan: to systematically determine the relationships between model-theoretic properties that relate to theory combination. An analysis of such properties can, for example, simplify proofs, in cases where a property follows from a combination of other properties.

In the first stage of this plan [10], we studied the relationships between all properties that relate to either polite or Nelson-Oppen combination, namely: stable infiniteness, smoothness, finite witnessability, strong finite witnessability, and convexity. The first two properties relate to the ability to enlarge cardinalities of models, while the next two require a computable *witness* function that restricts the models of a formula based on its variables. The last property relies on the ability to deduce an equality from a disjunction of equalities. The result of [10] was a comprehensive table: nearly every combination of these properties (e.g., theories that are smooth and stably infinite but do not admit the other properties) was either proved to be infeasible, or an example for it was given.

In this paper we continue with this plan by adding two properties: the finite model property and stable finiteness, both related to shiny theories [9]. The former requires finite models for satisfiable formulas, and the latter enforces bounds on them.

Of course, the theories from [10] can be reused. For these, one only needs to determine if they admit the finite model property and/or stable finiteness. The results and examples from [10] are, however, not enough. Given that the number of considered combinations is doubled with the addition of each property, new theories need to be introduced in order to exemplify the new possibilities, and new impossible combinations can be found. Hence, in this paper we provide several impossibility results for the aforementioned properties, as well as examples of theories for possible combinations. The overall result is a new table which extends that of [10] with two new columns corresponding to the finite model property and stable finiteness.[1]

The most interesting combinations that we study are theories that admit the finite model property but not finite witnessability. While both properties deal with finite models, the latter has a computable element to it, namely the witness function. In separating these properties, we found it useful to define theories that are based on the *Busy Beaver* function, a well known function from computability theory, that is not only non-computable, but also grows eventually faster than any computable function.

Outline: Sect. 2 reviews many-sorted logics and theory combination properties. Section 3 identifies combinations that are contradictory; Sect. 4 constructs the extended table of combinations, and describes the newly introduced theories. Section 5 gives final remarks and future directions this work can take. The proofs for the results in this paper may be found in an appendix to a preprint version of this work, available as [11].

2 Preliminary Notions

2.1 Many-Sorted Logic

A *many-sorted signature* Σ is a triple $(\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ where: $\mathcal{S}_\Sigma$ is a countable set of *sorts*; $\mathcal{F}_\Sigma$ is a countable set of function symbols; and $\mathcal{P}_\Sigma$ is a countable set of predicate

[1] While we use several results from [10], we do not assume here any familiarity with that paper. All required results are mentioned here explicitly.

symbols containing, for each $\sigma \in \mathcal{S}_\Sigma$, an equality $=_\sigma$. When σ is clear from the context, we write $=$. Every function symbol has an *arity* of the form $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, and every predicate symbol one of the form $\sigma_1 \times \cdots \times \sigma_n$, where $\sigma_1, \ldots, \sigma_n, \sigma \in \mathcal{S}_\Sigma$; equalities $=_\sigma$ have arity $\sigma \times \sigma$.

A signature that has no functions and only the equalities as predicates is called *empty*. Many-sorted signatures Σ where $\mathcal{S}_\Sigma$ has only one element are called *one-sorted*.

For any sort in $\mathcal{S}_\Sigma$ we assume a countably infinite set of variables, and distinct sorts have disjoint sets of variables; we then define first-order terms, formulas, and literals in the usual way. The set of free variables of sort σ in a formula φ is denoted by $vars_\sigma(\varphi)$, while $vars(\varphi)$ will denote $\bigcup_{\sigma \in \mathcal{S}_\Sigma} vars_\sigma(\varphi)$.

Σ-Structures $\mathbb{A}$ are defined as usual, by interpreting sorts (denoted by $\sigma^{\mathbb{A}}$), functions ($f^{\mathbb{A}}$) and predicate symbols ($P^{\mathbb{A}}$), with the restrictions that equality symbols are interpreted as identities. A Σ-interpretation $\mathcal{A}$ is an extension of a Σ-structure $\mathbb{A}$ with interpretations to variables. If $\mathbb{A}$ is the underlying Σ-structure of a Σ-interpretation $\mathcal{A}$, we say that $\mathcal{A}$ is an interpretation on $\mathbb{A}$. For simplicity, and because the use of structures is sparse in this paper, we will usually denote both structures and interpretations by using the same font, $\mathcal{A}$, $\mathcal{B}$ and so on. $\alpha^{\mathcal{A}}$ is the value taken by a Σ-term α in a Σ-interpretation $\mathcal{A}$, and if Γ is a set of terms, we simply write $\Gamma^{\mathcal{A}}$ for $\{\alpha^{\mathcal{A}} : \alpha \in \Gamma\}$.

We write $\mathcal{A} \vDash \varphi$ if the Σ-interpretation $\mathcal{A}$ satisfies the Σ-formula φ; φ is then said to be *satisfiable* if it is satisfied by some interpretation $\mathcal{A}$. The formulas found in Fig. 1 will be useful in the sequel. A Σ-interpretation $\mathcal{A}$: satisfies $\psi^\sigma_{\geq n}$ iff $|\sigma^{\mathcal{A}}| \geq n$; satisfies $\psi^\sigma_{\leq n}$ iff $|\sigma^{\mathcal{A}}| \leq n$; and satisfies $\psi^\sigma_{=n}$ iff $|\sigma^{\mathcal{A}}| = n$. For simplicity, when dealing with one-sorted signatures, we may drop the sort σ from the cardinality formulas.

$$\psi^\sigma_{\geq n} = \exists\, \overrightarrow{x}.\ \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j) \quad \psi^\sigma_{\leq n} = \exists\, \overrightarrow{x}.\, \forall\, y.\ \bigvee_{i=1}^{n} y = x_i \quad \psi^\sigma_{=n} = \psi^\sigma_{\geq n} \wedge \psi^\sigma_{\leq n}$$

Fig. 1. Cardinality Formulas. $\overrightarrow{x}$ stands for $x_1, \ldots, x_n$, all variables of sort σ.

A *Σ-theory* $\mathcal{T}$ is a class of all Σ-interpretations (called $\mathcal{T}$-interpretations) that satisfy some set $Ax(\mathcal{T})$ of closed formulas called the *axiomatization* of $\mathcal{T}$; the structures underlying these interpretations will be called the *models* of $\mathcal{T}$.

A formula is *$\mathcal{T}$-satisfiable* if it is satisfied by some $\mathcal{T}$-interpretation and, analogously, a set of formulas is $\mathcal{T}$-satisfiable if there is a $\mathcal{T}$-interpretation that satisfies all of them simultaneously. Two formulas are *$\mathcal{T}$-equivalent* when a $\mathcal{T}$-interpretation satisfies the first iff it satisfies the second. We write $\models_{\mathcal{T}} \varphi$, and say that φ is *$\mathcal{T}$-valid* if $\mathcal{A} \vDash \varphi$ for all $\mathcal{T}$-interpretations $\mathcal{A}$.

2.2 Theory Combination Properties

Let Σ be a signature, $\mathcal{T}$ a Σ-theory and $\mathcal{S} \subseteq \mathcal{S}_\Sigma$. We define several properties $\mathcal{T}$ may have with respect to S.

Convexity, Stable Infiniteness, and Smoothness $\mathcal{T}$ is *convex* with respect to S if for any conjunction of Σ-literals ϕ and any finite set of variables $\{u_1, v_1, \ldots, u_n, v_n\}$ of sorts in S with $\models_{\mathcal{T}} \phi \rightarrow \bigvee_{i=1}^{n} u_i = v_i$, one has $\models_{\mathcal{T}} \phi \rightarrow u_i = v_i$ for some i. $\mathcal{T}$ is *stably infinite* with respect to S if for every $\mathcal{T}$-satisfiable quantifier-free Σ-formula there is a $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying it such that $|\sigma^{\mathcal{A}}|$ is infinite for each $\sigma \in S$. $\mathcal{T}$ is *smooth* with respect to S if for every quantifier-free formula, $\mathcal{T}$-interpretation $\mathcal{A}$ that satisfies it, and function κ from S to the class of cardinals such that $\kappa(\sigma) \geq |\sigma^{\mathcal{A}}|$ for each $\sigma \in S$, there is a $\mathcal{T}$-interpretation $\mathcal{B}$ that satisfies it with $|\sigma^{\mathcal{B}}| = \kappa(\sigma)$ for each $\sigma \in S$.

(Strong) Finite witnessability For finite sets of variables V_σ of sort σ for each $\sigma \in S$, and equivalence relations E_σ on V_σ, the arrangement on $V = \bigcup_{\sigma \in S} V_\sigma$ induced by $E = \bigcup_{\sigma \in S} E_\sigma$, denoted by δ_V or δ_V^E, is the formula $\delta_V = \bigwedge_{\sigma \in S} \left[\bigwedge_{x E_\sigma y} (x = y) \wedge \bigwedge_{x \overline{E_\sigma} y} \neg(x = y) \right]$, where $\overline{E_\sigma}$ denotes the complement of the equivalence relation E_σ.

$\mathcal{T}$ is *finitely witnessable* with respect to S when there exists a computable function *wit*, called a *witness*, from the quantifier-free Σ-formulas to themselves that satisfies, for every ϕ: (i) ϕ and $\exists\, \overrightarrow{w}.\, \mathit{wit}(\phi)$ are $\mathcal{T}$-equivalent, for $\overrightarrow{w} = \mathit{vars}(\mathit{wit}(\phi)) \setminus \mathit{vars}(\phi)$; and (ii) if $\mathit{wit}(\phi)$ is $\mathcal{T}$-satisfiable, there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying $\mathit{wit}(\phi)$ such that $\sigma^{\mathcal{A}} = \mathit{vars}_\sigma(\mathit{wit}(\phi))^{\mathcal{A}}$ for each $\sigma \in S$.

Strong finite witnessability is defined similarly to finite witnessability, replacing (ii) by: $(ii)'$ given a finite set of variables V and an arrangement δ_V on V, if $\mathit{wit}(\phi) \wedge \delta_V$ is $\mathcal{T}$-satisfiable, there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ that satisfies $\mathit{wit}(\phi) \wedge \delta_V$ with $\sigma^{\mathcal{A}} = \mathit{vars}_\sigma(\mathit{wit}(\phi) \wedge \delta_V)^{\mathcal{A}}$ for all $\sigma \in S$. If $\mathcal{T}$ is smooth and (strongly) finitely witnessable with respect to $\mathcal{S}$, then it is *(strongly) polite* with respect to $\mathcal{S}$.

Finite Model Property and Stable Finiteness $\mathcal{T}$ has the *finite model property* with respect to S if for every quantifier-free $\mathcal{T}$-satisfiable Σ-formula, there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ that satisfies it with $|\sigma^{\mathcal{A}}|$ finite for each $\sigma \in S$. $\mathcal{T}$ is *stably finite* with respect to S if, for every quantifier-free Σ-formula and $\mathcal{T}$-interpretation $\mathcal{A}$ that satisfies it, there exists a $\mathcal{T}$-interpretation $\mathcal{B}$ that satisfies it with: $|\sigma^{\mathcal{B}}|$ finite for each $\sigma \in S$; and $|\sigma^{\mathcal{B}}| \leq |\sigma^{\mathcal{A}}|$ for each $\sigma \in S$. Clearly, stable finiteness implies the finite model property:

Theorem 1. *If $\mathcal{T}$ is stably finite w.r.t. S, then it has the finite model property w.r.t. S.*

We shall write **SI** for stably infinite; **SM** for smooth; **FW** (**SW**) for (strong) finitely witnessable; **CV** for convex; **FM** for the finite model property; and **SF** for stably finite.

3 Relationships Between Model-Theoretic Properties

In this section we study the connections between finiteness properties related to theory combination: the finite model property, stable finiteness, finite witnessability, and strong finite witnessability. We show how these properties are related to one another. In Sect. 3.1, we provide general results that hold for all signatures. Then, in Sect. 3.2, we focus on empty signatures, in which we are able to find more connections.

3.1 General Signatures

Finite witnessability, as well as its strong variant, were introduced in the context of polite theory combination. In contrast, the study of shiny theories utilizes the notions of

the finite model property, as well as stable finiteness. It was shown in [1] that for theories with a decidable quantifier-free satisfiability problem, shiny theories and strongly polite theories are one and the same. This already showed some connections between the aforementioned finiteness properties. However, that analysis also relied on smoothness, the decidability of the quantifier-free satisfiability problem of the studied theories, as well as the computability of the *mincard* function, the function that computes the minimal sizes of domains in models of a given formula in these theories.

Here we focus purely on the finiteness properties, and show that even without any other assumptions, they are closely related. Considering finite witnessability and the finite model property, notice that any witness ensures that some formulas always have finite models. Using the equivalence of the existential closure of such formulas to the formulas that are given to the witness, one gets the following result, according to which finite witnessability implies the finite model property.

Theorem 2. *Any Σ-theory $\mathcal{T}$ finitely witnessable with respect to $S \subseteq \mathcal{S}_\Sigma$ also has the finite model property with respect to S.*

Strong finite witnessability is a stronger property than finite witnessability, obtained by requiring finite models in the presence of arrangements. This requirement allows one to conclude stable finiteness for it, as the finer control on cardinalities that is required for stable finiteness can be achieved with the aid of arrangements. The following result is proved in Lemma 3.6 of [1], although under the assumption that the theory is smooth, something that is not actually used in their proof.

Theorem 3. *Any Σ-theory $\mathcal{T}$ strongly finitely witnessable with respect to $S \subseteq \mathcal{S}_\Sigma$ is also stably finite with respect to S.*

Clearly, stable finiteness implies the finite model property (Theorem 1). The converse does not generally hold, as we will see in Sect. 4. However, when these properties are considered with respect to a single sort, they actually coincide:

Theorem 4. *If a Σ-theory $\mathcal{T}$ has the finite model property with respect to a set of sorts S with $|S| = 1$, then $\mathcal{T}$ is also stably finite with respect to S.*

Theorems 2 and 3 are visualized in the Venn diagram of Fig. 2, where, for example, theories that are strongly finitely witnessable are clearly inside the intersection of finitely witnessable theories and stably finite theories.

When only one sort is considered, the picture is much simpler, and is described in Fig. 3. There, the finite model property and stable finiteness populate the same region, as ensured by Theorem 4. Notice that the results depicted in Fig. 3 hold for one-sorted and many-sorted signatures. The key thing is that the properties are all w.r.t. one of the sorts.

3.2 Empty Signatures

Figures 2 and 3 show a complete picture of the relationships between the properties studied in this section, for arbitrary signatures. However, when this generality is relaxed,

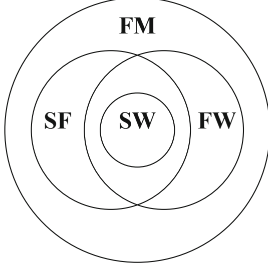


Fig. 2. Finiteness properties: general case.

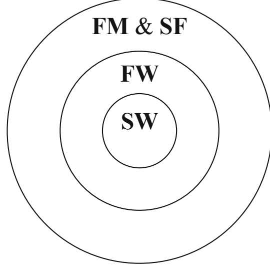


Fig. 3. Finiteness properties w.r.t. one sort.

several other connections appear. For this section, we require that the signatures are empty, and that they have a finite set of sorts. We further require that the properties in question hold for the entire set of sorts, not for any subset of it.

Table 1 defines the 5 signatures that will be used in the examples found in Sect. 4, and that will also appear in some of the results shown below: the empty signatures Σ_1, Σ_2 and Σ_3, with sets of sorts $\{\sigma\}$, $\{\sigma, \sigma_2\}$ and $\{\sigma, \sigma_2, \sigma_3\}$, respectively; and the signatures Σ_s and Σ_s^2 with one function s of arity $\sigma \to \sigma$, and sets of sorts $\{\sigma\}$ and $\{\sigma, \sigma_2\}$, respectively. Notice these are the simplest possible signatures when we order those by establishing: first, that the signature with fewer sorts is simpler; and second, that if two signatures have the same number of sorts, the one with fewer function symbols is simpler. We are free not to consider predicates, as they are at least as expressive as functions themselves; furthermore, we do not consider the problem of defining which of two signatures with the same numbers of sorts and function symbols is simpler, choosing rather to add only functions from a sort to itself.

Table 1. Signatures that will be used throughout the paper.

Signature	Sorts	Function Symbols
Σ_1	$\{\sigma\}$	$\emptyset$
Σ_2	$\{\sigma, \sigma_2\}$	$\emptyset$
Σ_3	$\{\sigma, \sigma_2, \sigma_3\}$	$\emptyset$
Σ_s	$\{\sigma\}$	$\{s : \sigma \to \sigma\}$
Σ_s^2	$\{\sigma, \sigma_2\}$	$\{s : \sigma \to \sigma\}$

First, in such a setting, we have that the finite model property implies finite witnessability, in the presence of smoothness.

Theorem 5. *If Σ is an empty signature with a finite set of sorts $\mathcal{S}_\Sigma$, and the Σ-theory $\mathcal{T}$ has the finite model property and is smooth with respect to $\mathcal{S}_\Sigma$, then $\mathcal{T}$ is also finitely witnessable with respect to $\mathcal{S}_\Sigma$.*

Next, we show that stable finiteness and smoothness together, imply strong finite witnessability.

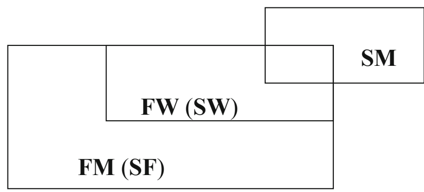


Fig. 4. Interplay between **SM**, **FW** (**SW**) and **FM** (**SF**) w.r.t. $\mathcal{S}_\Sigma$ in an empty signature.

Theorem 6. *If Σ is an empty signature with a finite set of sorts $\mathcal{S}_\Sigma$, and the Σ-theory $\mathcal{T}$ is stably finite and smooth with respect to $\mathcal{S}_\Sigma$, then $\mathcal{T}$ is also strongly finitely witnessable with respect to $\mathcal{S}_\Sigma$.*

While Theorem 2 and Theorem 3 establish certain unconditional relations between finite witnessability and the finite model property, and strong finite witnessability and stable finiteness, the converses shown to hold in Theorem 5 and Theorem 6 demand smoothness and that the properties hold with respect to the entire set of sorts. In that case, the situation can be represented by the diagram found in Fig. 4, showing clearly that a smooth theory that also has the finite model property (respectively, is stably finite), cannot not be finitely witnessable (strongly finitely witnessable).

Lastly, regarding the empty signatures Σ_1, Σ_2 and Σ_3, the following theorem shows that Σ_3 is sometimes necessary.

Theorem 7. *There are no Σ_1 or Σ_2-theories $\mathcal{T}$ that are, simultaneously, neither stably infinite nor stably finite, but are convex and have the finite model property, with respect to the entire set of their sorts.*

Hence, to exhibit such theories, one has to consider three-sorted theories.

4 A Taxonomy of Examples

In [10], we have created a table, in which for every possible combinations of properties from $\{$ **SI**, **SM**, **FW**, **SW**, **CV** $\}$ we either gave an example of a theory in this combination, or proved a theorem that shows there is no such example, with the exception of theories that are stably infinite and strongly finitely witnessable but not smooth. Such theories, referred to in [10] as *Unicorn Theories* (due to our conjecture that they do not exist) were left for future work, and are still left for future work, as the focus of the current paper is the integration of finiteness properties, namely **FM** and **SF** to the table.

And indeed, the goal of this section is to add two columns to the table from [10]: one for the finite model property and one for stable finiteness. The extended table is Table 2. We do not assume familiarity with [10], and describe the entire resulting table (though focusing on the new results).

Table 2. Summary of all possible combinations of theory properties. Red cells represent impossible combinations. In lines 26 and 34, $n > 1$; in lines 29, 30 and 35, $m > 1$, $n > 1$ and $|m - n| > 1$.

							Empty		Non-empty		
SI	**SM**	**FW**	**SW**	**CV**	**FM**	**SF**	One-sorted	Many-sorted	One-sorted	Many-sorted	$N^{\underline{o}}$
T	T	T	T	T	T	T	$\mathcal{T}_{\geq n}$	$(\mathcal{T}_{\geq n})^2$	$(\mathcal{T}_{\geq n})_s$	$((\mathcal{T}_{\geq n})^2)_s$	1
				F	T	T	[10]		$(\mathcal{T}_{\geq n})_\vee$	$((\mathcal{T}_{\geq n})^2)_\vee$	2
			F	T	T	T	Theorem 6		$\mathcal{T}_f$	$(\mathcal{T}_f)_s$	3
						F	Theorem 4	$\mathcal{T}_{2,3}$	Theorem 4	$(\mathcal{T}_{2,3})_s$	4
				F	T	T	[10]		$\mathcal{T}_f^s$	$(\mathcal{T}_f^s)^2$	5
						F			Theorem 4	$(\mathcal{T}_{2,3})_\vee$	6
		F	F	T	T	T	Theorem 5		$\mathcal{T}_\varsigma^s$	$\mathcal{T}_\varsigma^=$	7
						F			Theorem 4	$\mathcal{T}_\varsigma^2$	8
					F	F	$\mathcal{T}_\infty$	$(\mathcal{T}_\infty)^2$	$(\mathcal{T}_\infty)_s$	$((\mathcal{T}_\infty)^2)_s$	9
				F	T	T	[10]		$\mathcal{T}_\varsigma^\vee$	$(\mathcal{T}_\varsigma^\vee)^2$	10
						F			Theorem 4	$\mathcal{T}_{\varsigma\vee}^=$	11
					F	F			$(\mathcal{T}_\infty)_\vee$	$((\mathcal{T}_\infty)^2)_\vee$	12
	F	T	T	T	T	T	Unicorn				13
				F	T	T					14
			F	T	T	T	$\mathcal{T}^\infty_{even}$	$(\mathcal{T}^\infty_{even})^2$	$(\mathcal{T}^\infty_{even})_s$	$((\mathcal{T}^\infty_{even})^2)_s$	15
						F	Theorem 4	$\mathcal{T}^\infty$	Theorem 4	$(\mathcal{T}^\infty)_s$	16
				F	T	T	[10]		$(\mathcal{T}^\infty_{even})_\vee$	$((\mathcal{T}^\infty_{even})^2)_\vee$	17
						F			Theorem 4	$(\mathcal{T}^\infty)_\vee$	18
		F	F	T	T	T	$\mathcal{T}_\varsigma$	$(\mathcal{T}_\varsigma)^2$	$(\mathcal{T}_\varsigma)_s$	$((\mathcal{T}_\varsigma)^2)_s$	19
						F	Theorem 4	$\mathcal{T}_\varsigma^\infty$	Theorem 4	$(\mathcal{T}_\varsigma^\infty)_s$	20
					F	F	$\mathcal{T}_{n,\infty}$	$(\mathcal{T}_{n,\infty})^2$	$(\mathcal{T}_{n,\infty})_s$	$((\mathcal{T}_{n,\infty})^2)_s$	21
				F	T	T	[10]		$(\mathcal{T}_\varsigma)_\vee$	$((\mathcal{T}_\varsigma)^2)_\vee$	22
						F			Theorem 4	$(\mathcal{T}_\varsigma^\infty)_\vee$	23
					F	F			$(\mathcal{T}_{n,\infty})_\vee$	$((\mathcal{T}_{n,\infty})^2)_\vee$	24
F	F	T	T	T	T	T	$\mathcal{T}_{\leq 1}$	$(\mathcal{T}_{\leq 1})^2$	$(\mathcal{T}_{\leq 1})_s$	$((\mathcal{T}_{\leq 1})^2)_s$	25
				F	T	T	$\mathcal{T}_{\leq n}$	$(\mathcal{T}_{\leq n})^2$	$(\mathcal{T}_{\leq n})_s$	$((\mathcal{T}_{\leq n})^2)_s$	26
			F	T	T	T	[10]	$\mathcal{T}_1^{odd}$	$\mathcal{T}^{\neq}_{odd}$	$(\mathcal{T}_1^{odd})_s$	27
						F	Theorem 4	$\mathcal{T}^3_{2,3}$	Theorem 4	$\mathcal{T}^\infty_{\neq}$	28
				F	T	T	$\mathcal{T}_{\langle m,n\rangle}$	$(\mathcal{T}_{\langle m,n\rangle})^2$	$(\mathcal{T}_{\langle m,n\rangle})_s$	$((\mathcal{T}_{\langle m,n\rangle})^2)_s$	29
						F	Theorem 4	$\mathcal{T}^\infty_{m,n}$	Theorem 4	$(\mathcal{T}^\infty_{m,n})_s$	30
		F	F	T	T	T	[10]	$\mathcal{T}_1^\varsigma$	$\mathcal{T}^{\neq}_{\varsigma,1}$	$(\mathcal{T}_1^\varsigma)^2$	31
						F		$\mathcal{T}_\varsigma^{\infty,3}$	Theorem 4	$\mathcal{T}^\infty_{\varsigma\neq}$	32
					F	F		$\mathcal{T}_1^\infty$	$\mathcal{T}^{\neq}_{1,\infty}$	$(\mathcal{T}_1^\infty)_s$	33
				F	T	T		$\mathcal{T}_n^\varsigma$	$\mathcal{T}_\varsigma^{\neq}$	$(\mathcal{T}_n^\varsigma)_s$	34
						F		$\mathcal{T}^\varsigma_{m,n}$	Theorem 4	$(\mathcal{T}^\varsigma_{m,n})_s$	35
					F	F		$\mathcal{T}_2^\infty$	$\mathcal{T}^{\neq}_{2,\infty}$	$(\mathcal{T}_2^\infty)_s$	36

This section is structured as follows: In Sect. 4.1 we describe the structure of Table 2. In Sects. 4.2 to 4.4 we provide details about the axiomatizations of theories that populate it. Finally, in Sect. 4.5, we reuse operators from [10], prove that they preserve the finite model property and stable finiteness, and show how they are used in order to generate more theories for Table 2.

4.1 The Table

The columns left to the vertical double-line of Table 2 correspond to possible combinations of properties. In them, T means that the property holds, while F means that it does not. The first 5 columns correspond to properties already studied in [10], and the next two columns correspond to **FM** and **SF**. The columns right to the vertical double-line correspond to possible signatures: empty or non-empty, and one-sorted or many-sorted. White cells correspond to cases where a theory with the combination of properties induced by the row exists in a signature that is induced by the column. In such a case, the name of the theory is written. The theories themselves are defined in Figs. 5, 7 and 8, axiomatically. Shaded correspond to the cases where there is no such theory. In such a case, the theorem that excludes this possibility is written. If that theorem is from [10], we simply write [10].

Example 1. Line 1 of Table 2 corresponds to theories that admit all studied properties. We see that there is such a theory in each of the studied types of signatures (e.g., for the empty one-sorted signature, the theory $\mathcal{T}_{\geq n}$ exhibits all properties). In contrast, line 3 corresponds to theories that admit all properties but strong finite witnessability. We see that such theories exist in non-empty signatures, but not in empty signatures. This is thanks to Theorem 6.

Section 3, as well as results from [10], make some potential rows of Table 2 completely shaded. To allow this table to fit a single page, we chose to erase such rows. For example, by Theorem 1, there are no theories that are stably finite but do not have the finite model property, in any signature. Thus, no rows that represent such theories appear in the table.

In the remainder of this section, we describe the various theories that populate the cells of the table. Fortunately, all theories from [10] can be reused to exhibit also the new properties **SF** and **FM**, or their negations. These are described in Sect. 4.2. However, the theories from [10] alone are not enough. Hence we introduce several new theories in Sects. 4.3 and 4.4. Some of them are relatively simple, and are described in Sect. 4.3. Most of them, however, are more complex, and rely on the Busy Beaver function from theoretical computer science. We discuss these theories in Sect. 4.4.

4.2 Theories from [10]

For completeness, we include in Fig. 5 the axiomatizations of all theories from [10] that are used in Table 2 (Fig. 6 includes the definitions of formulas that are abbreviated in Fig. 5, such as $\psi^{=}_{\geq n}$ from the definition of $\mathcal{T}_f$). For lack of space, however, we refrain from elaborating on these theories, and refer the reader to their detailed description in [10]. For the theories of Fig. 5, whether they admit the properties from $\{\mathbf{SI}, \mathbf{SM}, \mathbf{FW}, \mathbf{SW}, \mathbf{CV}\}$ or not was already established in [10]. For each of them, here, we also check and prove whether they admit the new properties **FM** and **SF**.

Name	Sig.	Axiomatization
$\mathcal{T}_{\geq n}$	Σ_1	$\{\psi_{\geq n}\}$
$\mathcal{T}^{\infty}_{even}$	Σ_1	$\{\neg\psi_{=2k+1} : k \in \mathbb{N}\}$
$\mathcal{T}_{\infty}$	Σ_1	$\{\psi_{\geq k} : k \in \mathbb{N}\}$
$\mathcal{T}_{n,\infty}$	Σ_1	$\{\psi_{=n} \vee \psi_{\geq k} : k \in \mathbb{N}\}$
$\mathcal{T}_{\leq n}$	Σ_1	$\{\psi_{\leq n}\}$
$\mathcal{T}_{m,n}$	Σ_1	$\{\psi_{=m} \vee \psi_{=n}\}$

Name	Sig.	Axiomatization
$\mathcal{T}_{2,3}$	Σ_2	$\{(\psi^{\sigma}_{=2} \wedge \psi^{\sigma_2}_{\geq k}) \vee (\psi^{\sigma}_{\geq 3} \wedge \psi^{\sigma_2}_{\geq 3}) : k \in \mathbb{N}\}$
$\mathcal{T}^{\infty}_{2}$	Σ_2	$\{\psi^{\sigma}_{=2}\} \cup \{\psi^{\sigma_2}_{\geq k} : k \in \mathbb{N}\}$
$\mathcal{T}^{odd}_{1}$	Σ_2	$\{\psi^{\sigma}_{=1}\} \cup \{\neg\psi^{\sigma_2}_{=2k} : k \in \mathbb{N}\}$
$\mathcal{T}^{\infty}_{1}$	Σ_2	$\{\psi^{\sigma}_{=1}\} \cup \{\psi^{\sigma_2}_{\geq k} : k \in \mathbb{N}\}$

Name	Sig.	Axiomatization
$\mathcal{T}_f$	Σ_s	$\{[\psi^{=}_{\geq f_1(k)} \wedge \psi^{\neq}_{\geq f_0(k)}] \vee \bigvee_{i=1}^{k}[\psi^{=}_{=f_1(i)} \wedge \psi^{\neq}_{=f_0(i)}] : k \in \mathbb{N} \setminus \{0\}\}$
$\mathcal{T}^{s}_{f}$	Σ_s	$Ax(\mathcal{T}_f) \cup \{\psi_{\vee}\}$
$\mathcal{T}^{\neq}_{2,\infty}$	Σ_s	$\{[\psi_{=2} \wedge \forall\, x.\, p(x)] \vee [\psi_{\geq k} \wedge \forall\, x.\, \neg p(x)] : k \in \mathbb{N}\}$
$\mathcal{T}^{\neq}_{odd}$	Σ_s	$\{\psi_{=1} \vee [\neg\psi_{=2k} \wedge \forall\, x.\, \neg p(x)] : k \in \mathbb{N}\}$
$\mathcal{T}^{\neq}_{1,\infty}$	Σ_s	$\{\psi_{=1} \vee [\psi_{\geq k} \wedge \forall\, x.\, \neg p(x)] : k \in \mathbb{N}\}$

Fig. 5. Theories for Table 2 that were studied in [10]; $p(x)$ stands for $s(x) = x$. In $\mathcal{T}_f$, f is any non-computable function from the positive integers to $\{0, 1\}$, such that for every $k \geq 0$, f maps half of the numbers between 1 and 2^k to 1, and the other half to 0. In [10], such a function was proven to exist.

$$\psi^{=}_{\geq n} = \exists\, \overrightarrow{x}.\, \bigwedge_{i=1}^{n} p(x_i) \wedge \delta_n \qquad \psi^{=}_{=n} = \exists\, \overrightarrow{x}.\, [\bigwedge_{i=1}^{n} p(x_i) \wedge \delta_n \wedge \forall\, x.\, [p(x) \rightarrow \bigvee_{i=1}^{n} x = x_i]]$$

$$\psi^{\neq}_{\geq n} = \exists\, \overrightarrow{x}.\, \bigwedge_{i=1}^{n} \neg p(x_i) \wedge \delta_n \qquad \psi^{\neq}_{=n} = \exists\, \overrightarrow{x}.[\bigwedge_{i=1}^{n} \neg p(x_i) \wedge \delta_n \wedge \forall\, x.[\neg p(x) \rightarrow \bigvee_{i=1}^{n} x = x_i]]$$

$$\psi_{\vee} = \forall\, x.\, \left[(s(s(x)) = x) \vee (s(s(x)) = s(x))\right]$$

Fig. 6. Formulas for Σ_s-theories. $\overrightarrow{x}$ stands for $x_1, \ldots, x_n$. δ_n stands for $\bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$, and $p(x)$ stands for $s(x) = x$.

For example, for each n, $\mathcal{T}_{\geq n}$ consists of all Σ_1-structures that have at least n elements. This theory was shown in [10] to be strongly finitely witnessable, and so by Theorem 3 it is also stably finite. Then, by Theorem 1, it also admits the finite model property.

It is worth mentioning that $\mathcal{T}_{2,3}$ was first introduced in [1], in the context of shiny theories, where it was shown to have the finite model property, while not being stably finite. An alternative proof of this fact goes as follows: it was proven in [8] that $\mathcal{T}_{2,3}$ is: (i) finitely witnessable; (ii) not strongly finitely witnessable; and (iii) smooth. By

Name	Signature	Axiomatization
$\mathcal{T}^{\infty}$	Σ_2	$\{(\psi^{\sigma}_{=1} \wedge \psi^{\sigma_2}_{\geq k}) \vee diag^{\sigma,\sigma_2}(k+2) : k \in \mathbb{N}\}$
$\mathcal{T}^{\infty}_{m,n}$	Σ_2	$\{\psi^{\sigma}_{=\max\{m,n\}} \vee (\psi^{\sigma}_{=\min\{m,n\}} \wedge \psi^{\sigma_2}_{\geq k}) : k \in \mathbb{N}\}$
$\mathcal{T}^{\infty}_{\neq}$	Σ^2_s	$\{(\psi^{\sigma}_{=1} \wedge \psi^{\sigma_2}_{\geq k}) \vee (diag^{\sigma,\sigma_2}(k+2) \wedge \forall x.\, \neg p(x) : k \in \mathbb{N}\}$
$\mathcal{T}^{3}_{2,3}$	Σ_3	$\{\psi^{\sigma_3}_{=1}\} \cup \{(\psi^{\sigma}_{=2} \wedge \psi^{\sigma_2}_{\geq k}) \vee (\psi^{\sigma}_{\geq 3} \wedge \psi^{\sigma_2}_{\geq 3}) : k \in \mathbb{N}\}$

Fig. 7. Simple theories for Table 2. $diag^{\sigma,\sigma_2}(k+2)$, for any $k \in \mathbb{N}$, stands for the formula $(\psi^{\sigma}_{\geq k+2} \wedge \psi^{\sigma_2}_{\geq k+2}) \vee \bigvee_{i=2}^{k+2}(\psi^{\sigma}_{=i} \wedge \psi^{\sigma_2}_{=i})$, and $p(x)$ stands for $s(x) = x$.

Theorem 2 and (i), it also has the finite model property. But since it is over an empty signature, by (ii), (iii) and Theorem 6, we have that it cannot be stably finite.

4.3 New Theories: The Simple Cases

While the theories from Fig. 5 suffice to populate many cells of Table 2, they are not enough. Hence we describe new theories, not taken from [10]. The simplest theories that we have added can be found in Fig. 7, and are described below.

$\mathcal{T}^{\infty}$ is a theory with three distinct groups of models: its first group consists of models $\mathcal{A}$ that have $|\sigma^{\mathcal{A}}| = 1$ and $\sigma_2^{\mathcal{A}}$ infinite; its second group, of models $\mathcal{A}$ where both $\sigma^{\mathcal{A}}$ and $\sigma_2^{\mathcal{A}}$ are infinite; and its third group, of models $\mathcal{A}$ where $|\sigma^{\mathcal{A}}| = |\sigma_2^{\mathcal{A}}|$ is any value $k \geq 2$. In its axiomatization, one finds the formula $diag^{\sigma,\sigma_2}(k+2)$, equal to $(\psi^{\sigma}_{\geq k+2} \wedge \psi^{\sigma_2}_{\geq k+2}) \vee \bigvee_{i=2}^{k+2}(\psi^{\sigma}_{=i} \wedge \psi^{\sigma_2}_{=i})$ for $k \in \mathbb{N}$: that formula characterizes the models $\mathcal{A}$ of $\mathcal{T}^{\infty}$ that lie in the diagonal, that is, where $|\sigma^{\mathcal{A}}| = |\sigma_2^{\mathcal{A}}|$ (and this value is greater than 1), or both are infinite.

$\mathcal{T}^{\infty}_{m,n}$ is a theory that depends on two distinct positive integers m and n, and without loss of generality let us suppose $m > n$, when the theory has two types of models $\mathcal{A}$: in the first, $|\sigma^{\mathcal{A}}|$ equals m, while $\sigma_2^{\mathcal{A}}$ can be anything; in the second, $|\sigma^{\mathcal{A}}|$ equals n, and then $\sigma_2^{\mathcal{A}}$ must be infinite.

The models $\mathcal{A}$ of the Σ^2_s-theory $\mathcal{T}^{\infty}_{\neq}$ have either: $|\sigma^{\mathcal{A}}| = 1$, $|\sigma_2^{\mathcal{A}}| \geq \omega$ and $s^{\mathcal{A}}$ the identity function; both $\sigma^{\mathcal{A}}$ and $\sigma_2^{\mathcal{A}}$ infinite, and $s^{\mathcal{A}}$ with no fixed points; or $|\sigma^{\mathcal{A}}| = |\sigma_2^{\mathcal{A}}|$ equal to any number in $\mathbb{N} \setminus \{0, 1\}$, and again $s^{\mathcal{A}}$ with no fixed points.

Finally, $\mathcal{T}^{3}_{2,3}$ is made up of just the models $\mathcal{A}$ of $\mathcal{T}_{2,3}$ (see Fig. 5) with an extra domain associated to the new sort σ_3 such that $|\sigma_3^{\mathcal{A}}| = 1$.

4.4 New Theories: The Busy Beaver

So far we have seen that the theories from [10], together with a small set of simple new theories, can already get us quite far in filling Table 2. However, for several com-

Name	Signature	Axiomatization
$\mathcal{T}_\varsigma$	Σ_1	$\{\psi_{\geq\varsigma(k+2)} \vee \bigvee_{i=2}^{k+2} \psi_{=\varsigma(i)} : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^\infty$	Σ_2	$\{(\psi_{=1}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee diag_\varsigma^{\sigma,\sigma_2}(k+2) : k \in \mathbb{N}\}$
$\mathcal{T}_n^\varsigma$	Σ_2	$\{\psi_{=n}^{\sigma}\} \cup \{\psi_{\geq\varsigma(k+2)}^{\sigma_2} \vee \bigvee_{i=2}^{k+2} \psi_{=\varsigma(i)}^{\sigma_2} : k \in \mathbb{N}\}$
$\mathcal{T}_{m,n}^\varsigma$	Σ_2	$\{(\psi_n^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee (\psi_m^{\sigma} \wedge \psi_{\geq\varsigma(k+2)}^{\sigma_2}) \vee \bigvee_{i=2}^{k+2}(\psi_m^{\sigma} \wedge \psi_{=\varsigma(i)}^{\sigma_2}) : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^s$	Σ_s	$\{(\psi_{\geq k+1} \wedge \psi_{\geq\varsigma^{-1}(k+1)}^{=}) \vee \bigvee_{i=1}^{k+1}(\psi_{=i} \wedge \psi_{=\varsigma^{-1}(i)}^{=}) : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^{\neq}$	Σ_s	$\{(\psi_{=2} \wedge \forall x.\, \neg p(x)) \vee ((\psi_{\geq\varsigma(k+2)} \vee \bigvee_{i=2}^{k+2} \psi_{=\varsigma(i)}) \wedge \forall x.\, p(x)) : k \in \mathbb{N}\}$
$\mathcal{T}_{\varsigma,1}^{\neq}$	Σ_s	$\{\psi_{=1} \vee ((\psi_{\geq\varsigma(k+2)} \vee \bigvee_{i=2}^{k+2} \psi_{=\varsigma(i)}) \wedge \forall x.\, \neg p(x)) : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^{\vee}$	Σ_s	$\{\psi_\vee\} \cup \{(\psi_{\geq k+1} \wedge \psi_{\geq\varsigma^{-1}(k+1)}^{=}) \vee \bigvee_{i=1}^{k+1}(\psi_{=i} \wedge \psi_{=\varsigma^{-1}(i)}^{=}) : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^{=}$	Σ_s^2	$\{\psi_{\geq k+2}^{=} \rightarrow \psi_{\geq\varsigma(k+2)}^{\sigma_2} : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^2$	Σ_s^2	$\{(\psi_{=1}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee (\psi_{\geq k+2}^{=} \rightarrow \psi_{\geq\varsigma(k+2)}^{\sigma_2}) : k \in \mathbb{N}\}$
$\mathcal{T}_{\varsigma\vee}^{=}$	Σ_s^2	$\{\psi_\vee\} \cup \{(\psi_{=1}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee (\psi_{\geq k+2}^{=} \rightarrow \psi_{\geq\varsigma(k+2)}^{\sigma_2}) : k \in \mathbb{N}\}$
$\mathcal{T}_{\varsigma\neq}^{\infty}$	Σ_s^2	$\{(\psi_{=1}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee (diag_\varsigma^{\sigma,\sigma_2}(k+2) \wedge \forall x.\, \neg p(x)) : k \in \mathbb{N}\}$
$\mathcal{T}_\varsigma^{\infty,3}$	Σ_3	$\{\psi_{=1}^{\sigma_3}\} \cup \{(\psi_{=1}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee diag_\varsigma^{\sigma,\sigma_2}(k+2) : k \in \mathbb{N}\}$

Fig. 8. Busy Beaver Theories for Table 2. $diag_\varsigma^{\sigma,\sigma_2}(k+2)$ stands, for each $k \in \mathbb{N}$, for $(\psi_{\geq\varsigma(k+2)}^{\sigma} \wedge \psi_{\geq\varsigma(k+2)}^{\sigma_2}) \vee \bigvee_{i=2}^{k+2}(\psi_{=\varsigma(i)}^{\sigma} \wedge \psi_{=\varsigma(i)}^{\sigma_2})$, and $p(x)$ for $s(x) = x$; in $\mathcal{T}_{m,n}^\varsigma$, we assume w.l.g. $m \geq n$.

binations, it seems that more complex theories are needed. For this purpose, we utilize the well-known Busy Beaver function, and define various theories based on it. In this section, we describe these theories. First, in Sect. 4.4.1, we review the Busy Beaver function, and explain why it is useful in our context. Then, in Sects. 4.4.2 to 4.4.6, we describe the theories that make use of it, separated according to their signatures.

4.4.1 On the Busy Beaver Function The Busy Beaver function, here denoted ς, is an old acquaintance of theoretical computer scientists: essentially, given any $n \in \mathbb{N}$, $\varsigma(n)$ is the maximum number of 1's a Turing machine with at most n states can write to it's tape when it halts, if the tape is initialized to be all 0's. Somewhat confusingly, any Turing machine that achieves that number is also called a Busy Beaver.

It is possible to prove that $\varsigma(n) \in \mathbb{N}$ for any $n \in \mathbb{N}$ (see [5]), and so we may write $\varsigma : \mathbb{N} \rightarrow \mathbb{N}$; furthermore, ς is increasing. But the very desirable property of ς is that it is not only increasing, but actually very rapidly increasing.

More formally, Radó proved, in the seminal paper [5], that ς grows asymptotically faster than any computable function (being, therefore, non-computable). That is, for every computable function $f : \mathbb{N} \rightarrow \mathbb{N}$, there exists $N \in \mathbb{N}$ such that $\varsigma(n) > f(n)$ for all $n \geq N$. Despite that, the Busy Beaver starts somewhat slowly: $\varsigma(0) = 0$, $\varsigma(1) = 1$, $\varsigma(2) = 4$, $\varsigma(3) = 6$ and $\varsigma(4) = 13$; the exact value of $\varsigma(5)$ (and actually $\varsigma(n)$ for any $n \geq 5$) is not known, but is at least 4098 [3].

The fact that ς grows eventually faster than any computable function is a great property to have when constructing theories that admit the finite model property, while not being finitely witnessable. Roughly speaking, if the cardinalities of models of a theory are related to ς, this guarantees that it has models of sufficiently large finite size, while not being finitely witnessable since its models grow too fast: by carefully choosing formulas ϕ_n that hold only in the "n-th model" of the theory (when ordered by cardinality), the number of variables of $\mathit{wit}(\phi_n)$ offers an upper bound to $\varsigma(n)$ and is therefore not computable, leading to a contradiction with the fact that *wit* is supposed to be computable. Notice that, despite the dependency of our theories on the Busy Beaver, the function is not actually part of their signatures.

Now we present the theories that are based on ς. These theories are axiomatized in Fig. 8.

4.4.2 A $\boldsymbol{\Sigma_1}$-Theory The most basic Busy Beaver theory is $\mathcal{T}_\varsigma$. This is the Σ_1-theory whose models have cardinality $\varsigma(k)$, for some $k \geq 2$, or are infinite: that is, $\mathcal{T}_\varsigma$ has models with 4 elements, 6, 13 and so on. This theory forms the basis to all other theories of this section, that are designed to admit various properties from Table 2.

By itself, $\mathcal{T}_\varsigma$ has the finite model property while not being (strongly) finitely witnessable. It was in fact constructed precisely to exhibit this. As it turns out, it is also not smooth, but does satisfy all other properties. To populate other rows in the table that correspond to theories with other combinations of properties, more theories are needed, with richer signatures.

4.4.3 $\boldsymbol{\Sigma_2}$-Theories To fill the rows that correspond to other combinations, we introduce several Σ_2 theories.

The Σ_2-theory $\mathcal{T}_\varsigma^\infty$ is more complex. It has, essentially, three classes of models: the first is made up of structures $\mathcal{A}$ where $|\sigma^\mathcal{A}| = 1$ and $\sigma_2^\mathcal{A}$ is infinite; the second, of structures where both $\sigma^\mathcal{A}$ and $\sigma_2^\mathcal{A}$ are infinite; and the third, of structures where $|\sigma^\mathcal{A}| = |\sigma_2^\mathcal{A}|$ is a finite value that equals $\varsigma(k)$, for some $k \geq 2$. The formula $diag_\varsigma^{\sigma,\sigma_2}(k+2)$, for $k \geq 2$, in the axiomatization equals $(\psi^{\sigma}_{\geq\varsigma(k+2)} \wedge \psi^{\sigma_2}_{\geq\varsigma(k+2)}) \vee \bigvee_{i=2}^{k+2}(\psi^{\sigma}_{=\varsigma(i)} \wedge \psi^{\sigma_2}_{=\varsigma(i)})$ and is similar to $diag^{\sigma,\sigma_2}(k+2)$ from $\mathcal{T}^\infty$, characterizing the models $\mathcal{A}$ where either $|\sigma^\mathcal{A}| = |\sigma_2^\mathcal{A}| = \varsigma(k+2)$, or both $\sigma^\mathcal{A}$ and $\sigma_2^\mathcal{A}$ are infinite.

For each $n > 0$, $\mathcal{T}_n^\varsigma$ has as interpretations those $\mathcal{A}$ with $|\sigma^\mathcal{A}| = n$, and $|\sigma_2^\mathcal{A}|$either infinite or equal to $\varsigma(k)$, for some $k \geq 2$ (so $(|\sigma^\mathcal{A}|, |\sigma_2^\mathcal{A}|)$ may equal $(n, 4)$, $(n, 6)$, $(n, 13)$ and so on).

$\mathcal{T}_{m,n}^\varsigma$ is a Σ_2-theory that can be seen as some sort of combination of $\mathcal{T}_{m,n}^\infty$ and $\mathcal{T}_n^\varsigma$, dependent on two distinct positive integers m and n. Consider the case where the former is the greater of the two (the other cases are similar). In this case, we may divide its interpretations $\mathcal{A}$ into three classes: those with $|\sigma^\mathcal{A}| = n$ and $\sigma_2^\mathcal{A}$ infinite; those with $|\sigma^\mathcal{A}| = m$ and $\sigma_2^\mathcal{A}$ infinite; and those with $|\sigma^\mathcal{A}| = m$ and $|\sigma_2^\mathcal{A}|$ equal to some $\varsigma(k)$, for $k \geq 2$.

4.4.4 $\boldsymbol{\Sigma_s}$-Theories For some lines of Table 2, e.g. line 7, empty signatures are not enough for presenting examples. Hence we also introduce Σ_s-theories.

We start with $\mathcal{T}_\varsigma^s$, which is, arguably, the most confusing theory we here define: we are forced to appeal not only to the special cardinality formulas found in Fig. 6, but

also to the function ς^{-1}, which is a left inverse of ς. More formally, $\varsigma^{-1} : \mathbb{N} \to \mathbb{N}$ is the only function such that $\varsigma^{-1}(k) = \min\{l : \varsigma(l+1) > k\}$: so $\varsigma^{-1}(0) = 0$, $\varsigma^{-1}(1) = \varsigma^{-1}(2) = \varsigma^{-1}(3) = 1$, $\varsigma^{-1}(4) = \varsigma^{-1}(5) = 2$, $\varsigma^{-1}(6) = \cdots = \varsigma^{-1}(12) = 3$, $\varsigma^{-1}(13) = \cdots = \varsigma^{-1}(4097) = 4$, and further values of ς^{-1} are currently unknown. From the definition of ς^{-1}, we have that $\varsigma(\varsigma^{-1}(k)) \leq k$ and $\varsigma^{-1}(\varsigma(k)) = k$. ς^{-1} is not computable given that, since $\varsigma^{-1}(k) = \min\{l : \varsigma(l+1) > k\}$ by definition, $\varsigma^{-1}(k+1) \neq \varsigma^{-1}(k)$ iff $k+1$ is a value of ς: so, an algorithm to compute the values of ς could be obtained by simply computing the values of ς^{-1} and checking where there is a change.

$\mathcal{T}_{\varsigma}^{s}$ is then the Σ_s-theory with models $\mathcal{A}$ with any cardinality $k+1 \geq 1$, such that $s^{\mathcal{A}}(a) = a$ holds for precisely $\varsigma^{-1}(k+1)$ elements of $\mathcal{A}$, and so $s^{\mathcal{A}}(a) \neq a$ holds for $k+1-\varsigma^{-1}(k+1)$ elements, being the function $k \mapsto k+1-\varsigma^{-1}(k+1)$ itself non-decreasing, given that $\varsigma^{-1}(k+1)$ can equal either $\varsigma^{-1}(k)$ or $\varsigma^{-1}(k)+1$.

Example 2. We mention some $\mathcal{T}_{\varsigma}^{s}$-structures as examples: a structure $\mathcal{A}$ with $|\sigma^{\mathcal{A}}| = 1$ and $s^{\mathcal{A}}$ the identity; a structure $\mathcal{B}$ with $|\sigma^{\mathcal{B}}| = 2$ and $s^{\mathcal{A}}$ a constant function; a structure $\mathcal{C}$ with $|\sigma^{\mathcal{C}}| = 3$ (say $\sigma^{\mathcal{C}} = \{a, b, c\}$) and $s^{\mathcal{C}}$ the identity for only one of these elements (e.g., $s^{\mathcal{C}}$ can be a constant function, but now there are further possibilities such as $s^{\mathcal{C}}(a) = s^{\mathcal{C}}(b) = a$ and $s^{\mathcal{C}}(c) = b$); and a structure $\mathcal{D}$ with $|\sigma^{\mathcal{D}}| = 4$ (say $\sigma^{\mathcal{D}} = \{a, b, c, d\}$) and $s^{\mathcal{D}}$ the identity for only two of these elements (e.g., $s^{\mathcal{D}}(a) = s^{\mathcal{D}}(b) = s^{\mathcal{D}}(c) = a$ and $s^{\mathcal{D}}(d) = d$);

Next, we continue to describe other Σ_s theories.

$\mathcal{T}_{\varsigma}^{\neq}$ has essentially two classes of models $\mathcal{A}$: those with $|\sigma^{\mathcal{A}}| = 2$ and $s^{\mathcal{A}}$ never the identity; and those with $|\sigma^{\mathcal{A}}|$ equal to $\varsigma(k)$ or infinite, for some $k \geq 2$, and $s^{\mathcal{A}}$ the identity.

$\mathcal{T}_{\varsigma,1}^{\neq}$ is very similar to $\mathcal{T}_{\varsigma}^{\neq}$: the difference lies on where s will be the identity: while in $\mathcal{T}_{\varsigma}^{\neq}$ the function s is the identity for all interpretations $\mathcal{A}$ with $|\sigma^{\mathcal{A}}| > 2$, s in $\mathcal{T}_{\varsigma,1}^{\neq}$ is the identity only for the interpretations $\mathcal{A}$ with $|\sigma^{\mathcal{A}}| = 1$. So, in $\mathcal{T}_{\varsigma,1}^{\neq}$, we have a model $\mathcal{A}$ with $|\sigma^{\mathcal{A}}| = 1$ and $s^{\mathcal{A}}$ the identity, and then models $\mathcal{A}$ with $|\sigma^{\mathcal{A}}| = \varsigma(k)$ for some $k \geq 2$ or infinite, and $s^{\mathcal{A}}(a)$ anything but a.

The Σ_s-theory $\mathcal{T}_{\varsigma}^{\vee}$ is then just $\mathcal{T}_{\varsigma}^{s}$, satisfying in addition the formula $\psi_{\vee}$ (see Fig. 6). It has models $\mathcal{A}$ of any finite cardinality $k+1$, as long as $\varsigma^{-1}(k+1)$ of these elements a satisfy $s^{\mathcal{A}}(a) = a$, or infinite cardinalities, as long as the number of elements a satisfying $s^{\mathcal{A}}(a) = a$ is infinite; additionally, $s^{\mathcal{A}}(s^{\mathcal{A}}(a))$ must always equal either $s^{\mathcal{A}}(a)$ or a itself.

4.4.5 Σ_s^2-Theories

Now for theories in a many-sorted non-empty signature.

The Σ_s^2-theory $\mathcal{T}_{\varsigma}^{=}$ appears simple, but is actually quite tricky: starting by the easy case, if $\sigma^{\mathcal{A}}$ has infinitely many elements a satisfying $s^{\mathcal{A}}(a) = a$, $\sigma_2^{\mathcal{A}}$ is also infinite. If, however, the number of elements $a \in \sigma^{\mathcal{A}}$ satisfying $s^{\mathcal{A}}(a) = a$ is finite (notice that, even if this is the case, $\sigma^{\mathcal{A}}$ may still be infinite) and equal to some $k+2$, then $\sigma_2^{\mathcal{A}}$ has at least $\varsigma(k+2)$ elements. So, to give a better example, suppose $\sigma^{\mathcal{A}}$ has 2 elements satisfying $s^{\mathcal{A}}(a) = a$: then $\sigma_2^{\mathcal{A}}$ has at least $\varsigma(2) = 4$ elements, but may have any cardinality up to, and including, infinite ones; notice that in this example $\sigma^{\mathcal{A}}$ may be infinite as well, as long as only two of the elements satisfy $s^{\mathcal{A}}(a) = a$.

$\mathcal{T}_{\varsigma}^{2}$ is the same as $\mathcal{T}_{\varsigma}^{=}$, but with extra models $\mathcal{A}$ where $|\sigma^{\mathcal{A}}| = 1$ and $|\sigma_2^{\mathcal{A}}| \geq \omega$ (of course, then we have that $s^{\mathcal{A}}$ is the identity).

$\mathcal{T}_{\varsigma\vee}^{=}$ is then the same as $\mathcal{T}_{\varsigma}^{2}$, with the added validity of the formula $\psi_{\vee}$; So the models of $\mathcal{T}_{\varsigma\vee}^{=}$ are just models of $\mathcal{T}_{\varsigma}^{2}$ satisfying that $s^{\mathcal{A}}(s^{\mathcal{A}}(a))$ equals either $s^{\mathcal{A}}(a)$ or a itself.

$\mathcal{T}_{\varsigma\neq}^{\infty}$ is just the Σ_2-theory $\mathcal{T}_{\varsigma}^{\infty}$ with the added function s such that, if $|\sigma^{\mathcal{A}}| = 1$, $s^{\mathcal{A}}$ is the identity; and if $|\sigma^{\mathcal{A}}| > 1$, $s^{\mathcal{A}}(a)$ is anything but a.

4.4.6 A Σ_3-Theory Finally, $\mathcal{T}_{\varsigma}^{\infty,3}$ is obtained by adding a sort with a single element to the Σ_2-theory $\mathcal{T}_{\varsigma}^{\infty}$, similarly to the definition of $\mathcal{T}_{2,3}^{3}$, that was based on the Σ_2-theory $\mathcal{T}_{2,3}$ (see Sect. 4.3).

4.5 Theory Operators

There are two types of theories in Table 2: The first consists of base theories, such as $\mathcal{T}_{\geq n}$, that are axiomatized in Figs. 5, 7 and 8. The second is obtained from the first, by applying several operators on theories. For example, the theories $(\mathcal{T}_{\geq n})^2$, $(\mathcal{T}_{\geq n})_s$, $((\mathcal{T}_{\geq n})^2)_s$, are all obtained from the base theory $\mathcal{T}_{\geq n}$. So far we have only described the theories of the first type. In this section we explain the theories of the second type.

The operators that are used in Table 2 were defined in [10], in order to be able to systematically generate examples in various signatures. For example, if $\mathcal{T}$ is a Σ_1-theory, then $(\mathcal{T})^2$ is a Σ_2-theory with the same axiomatization as $\mathcal{T}$, that is, the second sort is completely free and is not axiomatized in any way. For completeness sake, we include the definitions of these operators here:

Definition 1 (Theory Operators from [10])

1. *If $\mathcal{T}$ is a Σ_1-theory, then $(\mathcal{T})^2$ is the Σ_2-theory axiomatized by Ax$(\mathcal{T})$.*
2. *Let Σ_n be an empty signature with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$, and let $\mathcal{T}$ be a Σ_n-theory. The signature Σ_s^n has sorts S and a single unary function symbol s of arity $\sigma_1 \to \sigma_1$, and $(\mathcal{T})_s$ is the Σ_s^n-theory axiomatized by Ax$(\mathcal{T}) \cup \{\forall x.\, [s(x) = x]\}$, where x is a variable of sort σ_1.*
3. *Let $\mathcal{T}$ be a theory over an empty signature with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$. Then $(\mathcal{T})_{\vee}$ is the Σ_s^n-theory axiomatized by Ax$(\mathcal{T}) \cup \{\psi_{\vee}\}$ (see Fig. 6).*

It was proven in [10] that these operators preserve the properties **SI**, **SM**, **FW**, **SW**, **CV**, and the lack of them. Here we prove that the same holds for **FM** and **SF** as well.

Theorem 8. *Let $\mathcal{T}$ be a Σ_1-theory. Then: $\mathcal{T}$ is **FM**, or **SF**, w.r.t. $\{\sigma\}$ if and only if $(\mathcal{T})^2$ is, respectively, **FM**, or **SF** w.r.t. $\{\sigma, \sigma_2\}$.*

Theorem 9. *If $\mathcal{T}$ is a theory over an empty signature Σ_n with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$, then: $\mathcal{T}$ is **FM**, or **SF**, w.r.t. S if and only if $(\mathcal{T})_s$ is, respectively, **FM**, or **SF**, w.r.t. S.*

Theorem 10. *If $\mathcal{T}$ is a theory over an empty signature Σ_n with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$, then: $\mathcal{T}$ is **FM**, or **SF**, w.r.t. S if and only if $(\mathcal{T})_{\vee}$ is, respectively, **FM**, or **SF**, w.r.t. S.*

Thus, in various cases, theories need not be invented from scratch, but can be generated from other theories. For example, the theory $\mathcal{T}_{\geq n}$ exhibits all studied properties, but is defined in a one-sorted signature. Using the operators, we obtain variants of this theory in all signature types, namely $(\mathcal{T}_{\geq n})^2$ for empty many-sorted signatures, $(\mathcal{T}_{\geq n})_s$ for non-empty one-sorted signatures, and $((\mathcal{T}_{\geq n})^2)_s$ for non-empty many-sorted signatures. The properties of the theories generated using these operators are guaranteed by Theorems 8 and 9, as well as the corresponding results from [10].

In two cases of theories defined using the Busy Beaver function, $\mathcal{T}_{\varsigma}^{\vee}$ and $\mathcal{T}_{\varsigma\vee}^{=}$, we cannot obtain them by relying on Theorem 10 from, respectively, $\mathcal{T}_{\varsigma}^{s}$ and $\mathcal{T}_{\varsigma}^{=}$, since the signatures of the latter theories are not empty. Curiously, adding $\psi_{\vee}$ to their axiomatizations still has the desirable outcome, but we prove this separately, without relying on Theorem 10. Extending Theorem 10 to non-empty signatures is left for future work.

The number of combinations of properties that we consider, together with the possible types of the signatures, adds up to $2^9 = 512$. Our negative results from Sect. 3 guarantee that only $\sim$15% of the actual table can be filled with examples. The remaining $\sim$85% are either shaded or are excluded from the table for space considerations. As for the examples that can be given, notice that there are in total an astonishing number of 78 theories in our table. But, thanks to the theory operators of Definition 1, only 33 of them ($\sim$42%) had to be concretely axiomatized in Figs. 5, 7 and 8. The remaining 45 theories were defined using the operators.

5 Conclusion

We examined, in addition to all properties considered in [10], the finite model property, and stable finiteness. Interesting restrictions for the combinations involving these properties were established. We also found interesting theories to fill in our table of combinations, most prominently those involving the Busy Beaver function as well as its inverse.

One possible direction this research could take is reasonably clear: considering the computability of the *mincard* function, what will, most probably, double the number of theories to be taken into consideration. Further interesting properties that could be considered include the decidability of the theory's axiomatization, or even its finiteness, and the satisfiability problem of the theory with respect to quantifier-free formulas.

Second, some of the negative results in [10] and in the present paper only hold with respect to the entire set of sorts $\mathcal{S}_{\Sigma}$. We plan to study if they hold also with respect to proper subsets of sorts, and if they do not, to provide counterexamples to those generalizations.

Acknowledgments. ([1]Funded in part by NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF) and ISF grant number 619/21.)

References

1. Casal, F., Rasga, J.: Many-sorted equivalence of shiny and strongly polite theories. J. Autom. Reason. **60**(2), 221–236 (2018)

2. Jovanović, D., Barrett, C.: Polite theories revisited. Technical report TR2010-922, Department of Computer Science, New York University, January 2010
3. Marxen, H., Buntrock, J.: Attacking the busy beaver 5. Bull. EATCS **40**, 247–251 (1990)
4. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
5. Radó, T.: On non-computable functions. The Bell Syst. Techn. J. **41**(3), 877–884 (1962)
6. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_3
7. Sheng, Y., Zohar, Y., Ringeissen, C., Lange, J., Fontaine, P., Barrett, C.W.: Polite combination of algebraic datatypes. J. Autom. Reason. **66**(3), 331–355 (2022)
8. Sheng, Y., Zohar, Y., Ringeissen, C., Reynolds, A., Barrett, C., Tinelli, C.: Politeness and stable infiniteness: stronger together. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 148–165. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_9
9. Tinelli, C., Zarba, C.: Combining decision procedures for theories in sorted logics. Technical report 04–01, Department of Computer Science, The University of Iowa, February 2004
10. Toledo, G.V., Zohar, Y., Barrett, C.: Combining combination properties: an analysis of stable infiniteness, convexity, and politeness. Accepted to CADE 2023 (2023). https://arxiv.org/abs/2305.02384
11. Toledo, G.V., Zohar, Y., Barrett, C.: Finite models and busy beavers, Combining finite combination properties (2023)

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.

Formal Reasoning Using Distributed Assertions

Farah Al Wardani, Kaustuv Chaudhuri(✉), and Dale Miller

Inria Saclay and LIX, Institut Polytechnique Paris, Palaiseau, France
{farah.al-wardani,kaustuv.chaudhuri,dale.miller}@inria.fr

Abstract. When a proof system checks a formal proof, we can say that its kernel *asserts* that the formula is a theorem in a particular logic. We describe a general framework in which such assertions can be made globally available so that any other proof assistant willing to trust the assertion's creator can use that assertion without rechecking any associated formal proof. This framework, called DAMF, is heterogeneous and allows each participant to decide which tools and operators they are willing to trust in order to accept external assertions. This framework can also be integrated into existing proof systems by making minor changes to the input and output subsystems of the prover. DAMF achieves a high level of distributivity using such off-the-shelf technologies as IPFS, IPLD, and public key cryptography. We illustrate the framework by describing an implemented tool for validating and publishing assertion objects and a modified version of the Abella theorem prover that can use and publish such assertions.

1 Introduction

In order to communicate a result from one formal reasoning system to another, a common technique is to transfer a formal proof certificate from the source system to the target system. This technique is usually required when the target system is *autarkic*,[1] wherein the system only trusts its own components, of which a particularly trusted component is an implementation of a proof checking *kernel*. To transfer a formal proof to an autarkic target system, either (a) the proof has to be *translated* from the source system, or (b) the verifier for the proof must be re-implemented as a *certified* procedure in the target system [6,25]. Both kinds of transferal are complicated for a variety of reasons: (1) The source and target system may not be syntactically, semantically, or foundationally compatible. (2) The source-proof language can have complex operational semantics that is cumbersome to encode in the target system. (Note that no universal standard has yet emerged for encoding the formal semantics of arbitrary proof languages; cf. Sect. 5.) (3) As systems change and mature, older versions of proof certificates can become stale and unmaintained. (4) Perhaps most importantly, many

[1] In [12], the adjective *autarkic* was applied to computational components of a proof checker but not to an entire proof checker.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 176–194, 2023.
https://doi.org/10.1007/978-3-031-43369-6_10

popular reasoning systems do not produce proof certificates at all. Prominent examples of that latter are SMT solvers that are not certifying when memory size and execution time are critical [32] and the specification tool Twelf [42] when using non-certifying procedures (e.g., totality checking).

Formal reasoning systems that are *non-autarkic* have an additional way to interact with external provers that addresses many of the above issues. In such systems, a host system is designed to build *proof obligations* that are then dispatched to external systems to solve. While these external systems may produce proofs, the host system usually does not check the proofs and instead *trusts the executions* of the external systems. This system architecture is most commonly used in program verification tools such as Dafny [28], Why3 [24], and TLAPS [16]. One issue not addressed with this enlarged view of trust is that the external dependencies tend to have unclear descriptions, especially from a third-party perspective. To illustrate, Dafny may declare that it trusts "Z3 v.4.12.1", but what does this mean? Is this external dependency to be interpreted by name, in which case any tool called "Z3 v.4.12.1" can be used, or is it precisely identified by, e.g., (a cryptographic hash of) the source code (or better, an executable binary) of a particular tool called "Z3 v.4.12.1"? Even with a precise identification, an external executable dependency may not be practical to incorporate. For example, the HOL Light system [27] re-checks its entire standard library every time it is started, taking on the order of minutes. If a development involves many calls to an external HOL Light-based solver, how are the calls to be orchestrated?

In addition to these two bases of trust—autarkic based on proof certificates, and non-autarkic based on executions of external tools—there is at least one other basis of trust in any heterogeneous development: the *agents* that write and assemble the developments and execute the formal tools as required (checkers, solvers, etc.). An example of an agent is a user, although one individual user can have many *agent profiles* (see Sect. 3.2). Entities such as a trustworthy central database can also correspond to an agent. Trusted agents have been largely neglected in the formal reasoning world, but they are common in other high reliability settings, such as security. Nevertheless, agents are at least implicitly present in any formal development: to claim that a result has been formally achieved is tantamount to saying that some trustworthy agent (e.g., peer reviewers) has *correctly and successfully* executed a specific collection of formal tools to convince themselves of that formal result. Furthermore, if one agent A *trusts* another B, there is no need for A to re-check B's proof scripts and re-execute any tools that B used to construct the result.

In this paper, we propose a framework where a *distributed* collection of agents can exchange formal results (called *assertions*), where the results have an unimpeachable *provenance*, and where each agent is in full control of their trust parameters. This *Distributed Assertion Management Framework* (DAMF) is:

- *Decentralized*: a global notion of *truth* is not imposed on every participant by means of a privileged logic, language, system, or software. This linguistic independence makes DAMF different from formalisms such as the *evidential tool bus* [20,38] that have been proposed for integrating external reasoning

agents into a unified formal system. Participants in DAMF are free to combine assertions from different sources if they believe the combination to be meaningful. Any participant can *retrieve* and use any assertion they understand, and this external import will be explicitly marked as a *dependency* if they choose to *publish* assertions they build with such external imports.
- *Reliable*: assertions have an *irrefutable* provenance, i.e., the fact that an agent has published an assertion is locally verifiable and independent of any other aspect of DAMF. Assertions, therefore, need to be *immutably* and *eternally* available, even in the presence of intermittent infrastructure and nefarious users or tools.
- *Composable*: assertions are not rigidly constrained by their *history*; new logical artifacts such as *theories*, *libraries*, *proof outlines*, etc. can be crafted by reorganizing existing assertions based on their declared dependencies.
- *Egalitarian*: the barrier to entry is low for participants who want to produce *or* consume such assertions.
- *Status Quo Compatible*: existing work already done with current mainstream systems is readily incorporated as assertions without needing to modify any existing system.

Concretely, DAMF provides JSON-based representations of a small number of concepts such as formulas, assertions, dependencies, etc. *without* any upfront commitment to a formal syntax or any particular semantics. These objects are then added to a *global store* in terms of the *InterPlanetary File System* (IPFS) [13] using linked data in the *InterPlanetary Linked Data* (IPLD) format. An object in IPFS/IPLD is denoted by a *canonical* content identifier (`cid`), a cryptographic hash of its content. Knowing the `cid` is sufficient to retrieve the object by any participant of the IPFS network. Furthermore, the `cid`s are the only externally visible *names* in DAMF, and links between objects are made using these `cid`s by IPLD. Features specific to a particular language or system, such as constants, variables, definitions, and notations, are kept localized to particular *formula objects*. Assertions are built using (the `cid`s of) formula objects and *signed* by their creator agents using public key cryptography. IPFS is used to distribute DAMF objects transparently using various technologies whose precise details are irrelevant to this paper.

This paper is accompanied by two concrete implementations that illustrate DAMF. First, we provide a tool called Dispatch that can be used by users and systems to both produce and consume DAMF assertions. Dispatch is not a privileged tool in DAMF: users and systems can interact directly with DAMF objects in IPFS if they so choose. Dispatch is simply one *interface* to the DAMF *global store*, making the integration of producers and consumers minimally demanding. It does tasks such as schematically validating the concrete JSON objects added to or retrieved from the global store. Dispatch also helps to analyze and modify the trust parameters for (compositions of) assertions.

Second, we implement a version of the Abella interactive theorem prover [10] that can produce and consume assertions in DAMF, mediated by Dispatch. As an example of its use, we show how Abella can use a lemma that was stated

and proved using the automated linear arithmetic reasoning tactics of Coq (v. 8.16.1); this lemma is manually translated from the Coq to the Abella language, with an explicit dependency on its Coq development, and added to the global store by the present authors. A user can accept this heterogeneous development as long as they trust Coq, Abella, and our translation of the Coq lemma to Abella. Moreover, this assertion, which contains explicit links to the externally sourced DAMF imports, can be published back to DAMF for use by others.

Since dependencies are explicitly tracked in DAMF assertions, any user can analyze various aspects of how it was composed of other assertions. Such analysis can form the basis of various kinds of *investigations*: for example, if a formula is found to be a non-theorem, an investigator can explore the compositions of the DAMF assertions that yield that formula in order to find the agents whose trust parameters may need to be modified. The Dispatch tool mentioned above comes with a command called *lookup* that explores combinations of known assertions that ultimately yield a desired result; for each such composition, the analysis extracts the collection of agents (and tools) that could be *trusted* in order to accept that composition.

In the next section, we describe the abstract design of DAMF and its underlying logic of assertions which form the basis of the abovementioned investigations. Section 3 describes our concrete implementation of DAMF, Sect. 4 discusses some of the design choices in DAMF, and Sect. 5 discusses some related work. The specific software tools (Dispatch and Abella-DAMF) accompanying this paper are fully documented at https://distributed-assertions.github.io/.

2 Design of DAMF

2.1 Languages, Contexts, and Formulas

To transfer a theorem from a source proof system to a target proof system, we must be able to transfer the statement of the theorem, which we represent as a *formula* object in DAMF. To be as general as possible, we represent the content of such a formula as a *string*, i.e., in a format suitable as an input to a parser of the source proof system. In order to determine that the input is well-formed, the source proof system may need further information about the *features*—symbols, predicates, functions, types, notations, hints, etc.—used in the formula. Such additional information is the *context* of the formula, which we represent as a document fragment in the language of the source proof system.

For example, take the following theorem written in `Coq 8.16.1`:

```
1 Definition lincomb (n j k : nat) := exists x y, n = x * j + y * k.
2 Theorem ex_coq : forall n:nat, 8 <= n -> lincomb n 3 5.
```

The formula corresponding to the theorem `ex_coq` is the literal string "`forall n:nat, ⋯ lincomb n 3 5`". The symbols `8`, `<=`, etc. are part of the standard prelude of this language, and the symbol `lincomb` is defined in line 1, so a sufficient context necessary for `Coq 8.16.1` to parse and type-check the theorem statement is the text of line 1, which is also written in the `Coq 8.16.1` language.

Abstractly, a *formula object* in DAMF is a triple (L, Σ, F) where L denotes a *language*, Σ denotes a *context*, and F denotes a *formula*, all of which may conceptually be thought of as strings. We will use the schematic variable N to range over such formula objects. The language L is a canonical identifier (specifically, the `cid` of a DAMF language object) which may optionally represent information about a suitable loader for the language that will make sense of the strings Σ and F; DAMF compares languages just by their identifiers. Moreover, L is interpreted as defining all the globally available features; for instance, the symbol `nat` is part of the standard prelude of this version of Coq and should therefore be understood as being defined in the language `Coq 8.16.1`. The context Σ introduces any user-defined features such as the definition `lincomb` above that is not part of Coq's standard prelude.

Note that DAMF formula objects are considered to be *closed*, i.e., every symbol used in the formula is defined in the language or the context. From the perspective of DAMF, a formula object is an atomic entity. Additionally, DAMF does not need to be aware of any reasoning principles of the language or context components. For instance, no mechanism in DAMF would allow the substitution of a declared symbol in the context with a concrete definition. The purpose of differentiating a formula object into three parts is purely pragmatic: the language part will in most cases be a well known object used by many agents, and the context part may potentially be shared between multiple assertions. DAMF consumers may be able to use this sharing of information to consolidate tasks such as context-processing.

2.2 Sequents and Assertions

A *sequent* in DAMF is abstractly of the form $N_1, \ldots, N_k \vdash N_0$ where each of the N_i is a DAMF formula object defined in the previous subsection. We will use the schematic variable Γ to range over ordered lists of formula objects, and S to range over sequents. In a sequent $\Gamma \vdash N$, we say that N is the *conclusion* and Γ are the *dependencies*. Such sequent objects may be produced whenever a formal proof has been checked in a proof checker: the conclusion represents the statement of the theorem, and the dependencies are external lemmas that were used during that proof. As an example, suppose the `Coq 8.16.1` theorem in Sect. 2.1 has a proof that appeals to the lemma `lem : forall m n, m <= n -> S m <= n \/ m = n`. The sequent that is produced is conceptually of the form `lem` $\vdash$ `ex_coq`, though concretely we would have to build DAMF formula objects by packaging the language and contexts.

An *agent* is a globally unique name. We use the schematic variable K to range over agents. We define a simple multi-sorted first-order logic where agents and sequents are primitive sorts and where the infix predicate *says* is the sole predicate; the atomic formula K *says* S, where K is an agent and S a sequent, is an *assertion*. The *says* predicate is implemented in DAMF using public-key cryptography. In a DAMF-aware proof system, when an appeal is made—say as part of the proof of some other theorem—to an assertion K *says* $(N_1, \ldots, N_k \vdash N_0)$, the appeal is interpreted as follows:

- The agent K is treated as *trusted*; if the agent cannot be trusted for some reason, such as if K occurs in a deny list, then the assertion is unusable.
- The conclusion of the assertion, N_0, contains the formula representing the lemma that is being appealed to. Note, in particular, that the dependencies $N_1, \ldots, N_k$ are not relevant to appealing to this assertion as an external dependency. These dependencies will be used in reasoning about compositions in DAMF, as described in Sect. 2.4.

2.3 Adapters

Because every formula object packages the formula together with its context and language identifier, every formula object is independent of every other formula object. Thus, in a sequent $N_1 \vdash N_0$, there is no requirement that the conclusion N_0 and the dependency N_1 be in the same language or have a common context. When working within a single autarkic system (e.g., a proof checker using a single logic), the sequents that are generated for every theorem will probably place the conclusion and dependencies in the same language and context; however, in the wider non-autarkic world, we can use multilingual sequents as first class entities that are documented and tracked the same way as any other kind of sequent.

An important class of multilingual sequents comes from *adapters*. In order for a theorem written in the `Coq 8.16.1` language to be used by a different system with a different language, say `Abella 2.0.9`, we will need to transform the formula objects in the former language to those in the latter language. This kind of translation is an example of a *language adapter*, which falls into the general class of *adapters*, and which creates a sequent by translating between languages or modifying the logical context by standard logical operations such as weakening (adding extra symbols), instantiation (replacing a symbol by a term), or unfolding (replacing a defined symbol by its definition).

As an example, the `Coq 8.16.1` example above can be translated to the `Abella 2.0.9` language as follows, where the function symbols `+` and `*` are replaced by relations in Abella.[2]

```
1 Import "nats". % some natural numbers library
2 Define lincomb : nat -> nat -> nat -> prop by
3   lincomb N J K := exists X Y U V,
4     times X J U /\ times Y K V /\ plus U V N.
5 Theorem ex_ab : forall n, nat n -> le 8 n -> lincomb n 3 5.
```

Lines 1–4 determine the context $\Sigma_{\texttt{ex_ab}}$ for the formula `ex_ab` on line 5.

The sequent that represents this translation therefore has the form

$$(\texttt{Coq 8.16.1}, \Sigma_{\texttt{ex_coq}}, \texttt{ex_coq}) \vdash (\texttt{Abella 2.0.9}, \Sigma_{\texttt{ex_ab}}, \texttt{ex_ab}).$$

Suppose agent K_1 signs this translation and that agent K_2 signs the sequent $\vdash (\texttt{Coq 8.16.1}, \Sigma_{\texttt{ex_coq}}, \texttt{ex_coq})$. As long as K_1 and K_2 are trusted by the user of `Abella 2.0.9`, then the formula object $(\texttt{Abella 2.0.9}, \Sigma_{\texttt{ex_ab}}, \texttt{ex_ab})$ can also be treated as a theorem by that user thanks to *composition*, discussed next.

[2] This encoding of functions using relations is the usual one: see [17] for details.

2.4 Composing Assertions, Trust

Assertions will be composed by means of a single rule of inference that implements a cut-like rule for sequents, Compose.

$$\frac{K\ \textit{says}\ (\Gamma_1 \vdash M) \qquad K\ \textit{says}\ (M, \Gamma_2 \vdash N)}{K\ \textit{says}\ (\Gamma_1, \Gamma_2 \vdash N)}\ \textsc{Compose}$$

The effect of this rule means that the *says* predicate does not correspond one-to-one with cryptographic signatures. The conclusion of the Compose rule may, in particular, not be a sequent that has been explicitly signed by the agent K even if both premises are. Rather, the rule states that whenever K can be said to reliably claim, *either* by a cryptographic signature *or* by a Compose-derivation tree, that both $\Gamma_1 \vdash M$ and $M, \Gamma_2 \vdash N$, then K must also reliably claim $\Gamma_1, \Gamma_2 \vdash N$.

There are many variations to *access control logic* in the literature. For example, some such logics use inference rules such as:

$$\frac{\Gamma \vdash N}{K\ \textit{says}\ (\Gamma \vdash N)} \quad \text{or} \quad \frac{K\ \textit{says}\ (\Gamma \vdash N)}{K\ \textit{says}\ (K\ \textit{says}\ (\Gamma \vdash N))}.$$

Such rules are neither syntactically well-formed nor desirable for our purposes. We use here a very weak access control logic (see [1] for a survey of such logics). Instead, checking the validity of a given derivation using Compose is computationally trivial: each instance of it must eliminate exactly the leftmost dependency in the second premise, which is a DAMF formula object that is compared by `cid`.

Observe that the agent K does not participate in a meaningful way in a derivation that is built with the Compose rule. Thus, for a given end sequent of the form K *says* $(\vdash N)$, a Compose derivation can be seen as a *proof outline* for the desired theorem N, with the leaves of the derivation being the assertions that need to be sourced from an assertion database (such as the DAMF global store). We say that an assertion (K *says* S) is *published* if it can be retrieved from such a database. The inference system is then enlarged with the following rule that can be used to complete the open leaves of the Compose derivation using assertions made by different agents.

$$\frac{(K_1\ \textit{says}\ S)\ \text{is published}}{K_2\ \textit{says}\ S}\ \textsc{Trust}\ [K_1 \mapsto K_2]$$

This rule is parameterized by a pair of agents, K_1 and K_2, and is understood to be applicable only when K_1 is in the user-specified *allow list* of K_2 (i.e., K_1 *speaks for* K_2, which we write as $[K_1 \mapsto K_2]$).

We do not assume that agents have any additional closure properties beyond Compose and Trust. For example, suppose N_A, $N_{A \to B}$, and N_B are the formula objects that correspond to the formulas A, $A \to B$, and B respectively in

some language. We do not assume that the following rule is admissible:

$$\frac{K\ \textit{says}\ (\Gamma \vdash N_{A \to B}) \qquad K\ \textit{says}\ (\Gamma \vdash N_A)}{K\ \textit{says}\ (\Gamma \vdash N_B)}\ \text{MP}.$$

That is, we do not assume that the formulas asserted by agent K are closed under modus ponens. Similarly, we do not assume that what agents assert are closed by substitution or instantiation of any symbols that are defined in the contexts of the formula objects. While a particular agent may not be closed under modus ponens, substitution, or instantiation, it is possible to employ other agents that can look for opportunities to apply such inference rules on the results of trusted agents. In particular, if we want the query engine to be able to use the MP rule, then the engine must construct an agent K_{MP} whose sole function is to generate assertions such as K_{MP} *says* $(N_{A \to B}, N_A \vdash N_B)$ that correspond to applications of the MP rule. Of course, K_{MP} will need to be in the *allow list* for any agent wanting to use this agent.

2.5 Producing Assertions, Formal Reasoning Tools

Conceptually, an agent constructs a DAMF sequent as a consequence of running formal reasoning tools such as proof checkers or theorem provers. DAMF includes *tool objects*, which are unconstrained JSON objects that can be used to describe such tools. A *tool object* does not necessarily describe an implemented tool; it might describe a part of it, or an abstract description of the logical system in which the sequent is asserted in, for instance. Like with languages in Sect. 2.1, we compare tools for equality by means of the `cid`s of these tool objects. It is also possible for an agent to build a DAMF sequent manually, without running any tool. The agent may do this for a number of reasons: e.g., the assertion may be a *conjecture* (i.e., a proof may be provided at some other time but is currently missing) or a manually produced *adapter*.

A DAMF *production* is a sequent that is annotated with a *mode* that describes how the sequent was produced; this mode can be the `cid` of a tool object mentioned above, or it can be *null* expressing an *unproven* sequent. We use the schematic variable T for modes, and write a production of the sequent $\Gamma \vdash N$ with mode T as $\Gamma \vdash_T N$. Published DAMF assertions will be of the form K *says* $(\Gamma \vdash_T N)$, and we modify the TRUST rule to the following:

$$\frac{(K_1\ \textit{says}\ (\Gamma \vdash_T N))\ \text{is published}}{K_2\ \textit{says}\ (\Gamma \vdash N)}\ \text{TRUST}\ [K_1/T \mapsto K_2]$$

where the side condition $[K_1/T \mapsto K_2]$ means that K_2 allows K_1's assertions in mode T. It may be tempting to think of K_1/T as an agent by itself, but, as we shall see in Sect. 3.1, agents are implemented in DAMF using keypairs, so if K_1/T_1 and K_1/T_2 were separate agents then there would be no verifiable way to link them both to K_1. This use of modes makes it possible, for example, to trust an agent K using any version of Coq while not trusting K when using other proof systems.

2.6 Logical Consistency of Heterogeneous Combinations

DAMF imposes no constraints on the composition of assertions, which can at first glance appear to be risky. For example, suppose the assertions come from incompatible logics, say an assertion in classical logic during the proof of an intuitionistic theorem. Without exceptional care, the result of a COMPOSE will only be classically, not intuitionistically, true. Similar problems exist if the imported assertion requires additional axioms that are incompatible with the user's setting (e.g. extensionality or UIP in the setting of univalence).

This issue highlights the fact that DAMF *does not* guarantee logical compatibility of assertions; rather, DAMF is more accurately seen as a *record* of compositions that have been made. To trust an agent's assertion is just to say that we trust that the agent indeed had good reasons (such as a proof) to make that assertion, *not* that the assertion may be arbitrarily composed. Moreover, DAMF assertions are intended to be read as *hypothetical statements* from dependencies to conclusions (where "*hypothetical*" is understood in the informal language of discourse rather than as a formal implication or entailment). If the dependencies cannot be met, the assertion is useless. To illustrate, if an agent K wants to use an assertion $\Gamma \vdash M$ in their proof of N, the assertion they will publish is K *says* $(M \vdash N)$, which is acceptable in isolation; if M is incompatible with the logic of N, then the assertion K *says* $(M \vdash N)$ is vacuous.

3 Implementation: Information, Processes, and Tools

3.1 The Structures of the Global Store

A crucial design criterion of DAMF is that the assertions and their constituent objects are a globally shared commodity, existing independently of the tools that produce or consume them. To this end, DAMF requires well-defined basic structures that producers would produce and consumers would expect and know how to address.

The use of a content-addressing scheme is an essential part of seeing these structures as global. Each structure is identified and addressed by a unique global identifier in a common namespace in an independently verifiable and trusted way: the identifier is derived from the content itself and every alteration of the content produces a new identifier; at the DAMF level, *the content is the name/address*, and comparing two objects structurally at the DAMF level is reduced to comparing their `cids` as strings. One way to handle differences in `cids` between different forms of conceptually the same DAMF object is by curation and normalization of such structures at the level of producers or potentially other DAMF actors.

The structures we may want to specify in DAMF are built by composing several elements; for instance, a *sequent* contains *formula* structures, which themselves contain *context* structures. In DAMF, we make the design choice to treat all such structures as *first class* objects stored in a distributed network through

IPFS, and use the linked data representation of IPLD to represent an object as being composed of other objects.

The core DAMF structures we define are *context*, *formula*, *sequent*, *production*, and *assertion*. Concretely, these structures are represented as JSON objects with a varying `format` property which has the type of the structure as its value. These structures are described as follows (full definitions in [4, Appendix A]):

- *Context*: contains a `language` field, which is an IPLD link to a *language* object, described in Sect. 2.1, and a `content` field containing the body of the context.
- *Formula*: contains a `language` field, a `content` field for a string representation of the formula in the language, and a `context` field that is an IPLD link to a context object, as described in Sect. 2.1.
- *Sequent*: a `dependencies` field mapped to a list of IPLD links to formula objects, and a `conclusion` field as an IPLD link to a formula object.
- *Production*: pairs a sequent object with a `mode` field denoting a *mode of production* of a sequent as described in Sect. 2.5.
- *Assertion*: a `claim` field mapped to an IPLD link to a production (currently considered the main claim type in DAMF), an `agent` field mapped to a public key, and a `signature` field containing the result of signing the `cid` of the value of the `claim` field.

Given these schemata, the aspects of tracking and trusting become natural: a formula present as a `dependency` in some assertion could be matched with the same formula present as the `conclusion` of a different assertion.

It is also useful to annotate these core DAMF objects with additional metadata such as external names, proof objects, timestamps, etc. In DAMF, we have chosen to give the core objects a `cid` independent of the metadata; instead, for every core object, we define an *annotated* object that is composed of a link to the core object and a link to any additional metadata. DAMF follows the design principle that objects are to be considered equal at the DAMF level if they have the same `cid`: the content of the objects is not examined, and no IPLD-links are followed for such comparisons. Generally speaking, therefore, DAMF core objects will not link to annotated objects, since the annotations will factor into the `cid`s and force disequality when undesired, such as when building compositions (Sect. 2.4). The sole exception to this rule of thumb are assertion objects which can use annotated production objects as their claims. Note that every assertion object will be globally unique when produced: it will have a different `cid` each time its claim is signed, even if signed by the same agent, because cryptographic signatures always include a nonce.

Another layer of structures that can aggregate global object references are *collections*. We currently define one generic *collection* format in our implementation: many other non-generic collection formats can easily be considered.

3.2 Processes in DAMF, and Dispatch as an Intermediary Tool

The two obvious processes in DAMF are the *production* and *consumption* of DAMF objects. In a *production* process, DAMF objects are constructed starting

from local information, published, and then stored across the distributed network. The *consumption* process is in the opposite direction: locally consumable information are constructed from DAMF objects. The important point is that these DAMF objects are common and well-understood (as DAMF formats) for all consumers, and each consumer decides what to consume and how to consume it. For example, a consumer might only choose to read formulas that are of some specific language, and then decide how to process their internal structures based on its own criteria. Other than these two, other processes will be done on the published DAMF objects that will incorporate their combination, curation, and analysis. The process we consider first in our implementation is *lookup* which will be discussed further below. Individual producers and consumers, such as theorem provers, can choose to implement some or several of these DAMF processes. However, many aspects of dealing with linked data and IPFS will be common to such tools, so we describe an intermediary tool called Dispatch that simplifies the interactions between these producers and consumers and the DAMF global store. Of course, Dispatch would be considered part of the *trusted code base*, along with IPFS and any utilities used to manipulate JSON data and cryptographic signatures. If this is problematic, Dispatch can be completely foregone in preference to native implementations.

The Dispatch tool is distributed as an executable `dispatch` with three subcommands: `publish`, `get`, and `lookup`. The `dispatch publish` command operates on one of a collection of standard input formats that contains local information corresponding to DAMF types. After syntactically validating this input, the `publish` command will construct and publish the global objects. Dispatch can also optionally interact with a specific storage service in order to make that object widely discoverable in the IPFS network. As an example, consider the following input for an *assertion* object, where newly created formulas and contexts are placed in the same file and are referred by local names such as `plus_comm`, and previously existing objects are referred by their `cids` using the `damf:` flag, such as the first value of "`dependencies`" (line 10) which refers to a *formula* object `cid`, as well as "`language`" and "`mode`" values which refer to existing *language* and *tool* objects respectively.

```
1  { "format": "assertion",
2    "agent": "localAgent",
3    "claim": {
4      "format": "annotated-production",
5      "annotation": ...,
6      "production": {
7        "mode": "damf:bafyreihnx2...",
8        "sequent": {
9          "conclusion": "plus_comm",
10         "dependencies": [ "damf:bafyreihw6g...", "plus_succ" ] } } },
11   "formulas": {
12     "plus_comm": {
13       "language": "damf:bafyreidyts...",
14       "content": ": forall M N K, nat K -> ...",
```

```
15        "context": ["plus"] },
16      "plus_succ": {
17        "language": "damf:bafyreidyts.....",
18        "content": ": forall M N K, ...",
19        "context": ["plus"] } },
20    "contexts": {
21      "plus": {
22        "language": "damf:bafyreidyts.....",
23        "content": [
24            "Kind nat type.", "Type z nat.", "Type s nat -> nat.",
25            "Define plus : nat -> nat -> prop by ...." ] } } }
```

This example is based on an output from our Abella-DAMF prover described below. A prover using Dispatch tool only needs to be able to produce and consume JSON objects with this structure, without needing to interface with IPFS directly. The value of "`agent`" (line 2) refers to an *agent profile* in Dispatch; each profile maps a user-readable name to a cryptographic key-pair, created separately using the `dispatch create-agent` command.

The `dispatch get` command takes a `cid` as an argument, fetches the IPLD `dag` (the full JSON object) referenced by it from the global store, validates the types of all constituent IPLD linked objects, verifies any signatures, and finally outputs a JSON object that is similar in structure to that accepted by `dispatch publish`. The consumer will have access to all the necessary DAMF objects referenced by the root `cid` without needing to interact with the global store or structurally validating any objects. The only difference between the output of `dispatch get` and the input of `dispatch publish` is that the local names that appeared in the input will be replaced by `cids` (i.e., *global names*) in the output. Input and output formats corresponding to other global types are described further at the site mentioned in the introduction.[3]

The `dispatch lookup` command, as mentioned earlier, is the starting process that we consider in our implementation regarding the combination and analysis of DAMF assertions. Given a formula `cid` and a collection of assertion `cids`, the output of this command is a list of potential sets of (agent, mode/tool) pairs that correspond to combinations of assertions that would yield the target formula. Any remaining unmatched dependency is also outputted along with the (agent, mode/tool) pairs. In our current implementation, Dispatch exhaustively generates all possible ways of constructing the target formula. A direct improvement is to change this aspect of the tool to allow for a more interactive and incremental exploration of such dependencies. In addition, filtering through allow-lists would reduce the number of assertion combinations generated by this command.

3.3 Edge Systems Example: Abella

We have implemented a DAMF-aware branch of Abella [10] as an example of a system that interacts with assertions in DAMF with the help of Dispatch as a

[3] https://distributed-assertions.github.io/.

mediator. Abella was originally designed to test a particular approach to meta-theoretic reasoning using a new, proof-theoretically motivated mechanism for reasoning directly with bound variables (in particular, the ∇-quantifier [30] and a treatment of equality based on equivariant higher-order unification [26]). While the current implementation of Abella has succeeded with those meta-theoretic tasks [22,41], the prover has not grown much beyond that domain. Indeed, Abella has some (mis)features that make it a good test case for DAMF: (1) it has no awareness of the file system and it is easy to replace the backing store from local files to objects stored in IPFS; (2) it has a feature-poor proof language with nearly no support for proof automation and hence an underdeveloped formal mathematical libraries; and (3) it uses *relational* specifications as opposed to the more common *functional programming* specifications. Furthermore, the area of meta-theory that Abella treats declaratively is also an area many conventional proof systems do not deal well, in part, because of the need to encode and manipulate bindings [9,23]. Such conventional systems might be willing to delegate such meta-theoretic reasoning to Abella.

Ordinary Abella developments (in `.thm` files) support a kind of *import* mechanism which loads in marshaled results from a different run of Abella. We extend *import* with a new kind of statement: `Import` "`damf:bafyr`..." that refers to a collection of DAMF assertions (i.e., a DAMF collection object whose elements are assertions). Dispatch is used to fetch all the referenced objects from IPFS as explained in the previous subsection.

To appeal to an assertion, the elements of the context of the conclusion of the assertion are *merged* using their internal names with the ambient context of Abella where the assertion is appealed to. An Abella declaration in the context is *mergeable* if it has both the same internal name and an identical (up to λ-equivalence) definition; thus, type and term constants are merged if they have the same kinds or types (respectively), and (co-)definitions are merged if they have the same definitional clauses. This is done to keep the implementation simple and mostly unchanged from the standard (non-DAMF) Abella, which also only allows an `Import` declaration when the imported objects can be merged.

When the proof of a theorem is completed in Abella, a sequent object is constructed with the dependencies being all the DAMF lemmas appealed to in the proof, and the conclusion being the statement of the theorem (the formula) in the context of all its necessary declarations, computed using a dependency analysis. We use only the necessary declarations to allow such DAMF sequents to have the widest possible uses, since a DAMF assertion can only be used in Abella if the *entire* context of the conclusion can be merged.

A full example of an Abella development that makes use of imported assertions from Abella, Coq, and λProlog can be found in [4, Appendix B]. In this example, Coq and λProlog are not modified at all, and Abella is only minimally modified to use Dispatch to interact with DAMF assertions. The total amount of modifications to Abella to interface with Dispatch amounts to about 100 lines of code, most of which deals with (un)marshalling JSON. We expect that making tools DAMF-aware would require negligible effort.

4 Discussion: Design Choices and Alternatives

4.1 The Role of Formal Proofs

Autarkic theorem provers often exploit the existence of proofs for several reasons. Obviously, the ability to check a fully detailed proof object in their own kernel, following the *De Bruijn criterion* [11], is central. But proofs can also be used for various other roles. For example, they sometimes contain constructive content that can be extracted as executable programs, and they can be used as guides during the development and maintenance of other proofs. Given their central role in many proof assistants, a great deal of effort has gone into the formalization, manipulation, and transformation of formal proof objects; see, for example, MMT [35], Logipedia [21], and foundational proof certificates [18]. As a concrete matter, proof objects can be included in the annotations of annotated productions in the global store of DAMF. Sequents are linked in productions by their `cids`, so it is possible for the same sequent to have multiple proof objects contributed by different agents in separate assertions.

4.2 Potential Benefits to Mainstream Systems

The fact that proof objects are not central to DAMF and the example presented in Sect. 3.3 might lead the reader to believe that the only beneficiaries of DAMF are new systems that want to leverage existing developments in mainstream systems. This belief is not necessarily true for two reasons. First, there are certain logical systems and formalization styles that are inordinately complicated or impossible to do in mainstream systems. Good examples are nominal sets [34], λ-tree syntax (a.k.a. *higher-order abstract syntax*) [2,23], generic judgments [30], and nominal abstraction [26]. It is conceivable that a mainstream prover can use DAMF to import a formalization such as the proof of soundness of Howe's method done in the setting of higher-order abstract syntax and contextual modal type theory [31], which is at present not available in a mainstream proof system such as Coq or Agda.

A second benefit to mainstream systems is to enable more trustworthy refactoring of their existing implementations. For example, modern autarkic provers routinely recheck large collections of proofs, often after every invocation of a new instance of the proof checker and certainly after every change in the version of the prover. As a result of needing to recheck such proofs, there is a tendency for implementers of proof checkers to optimize such kernels to be more efficient. However, such optimizations can add greater complexity to a kernel, making errors in the kernel more likely to occur. With DAMF, once a trustworthy but slow kernel—e.g., a certified implementation of a kernel [39]—checks a proof, it rarely needs to be rechecked. This can even lower the pressure for kernel implementations to chase performance with increasing, error-prone complexity. Furthermore, the immutable nature of IPFS objects makes DAMF assertions resistant to malicious subversion of the proper execution of a tool – see, for example, the discussion in [5] concerning attacks on Coq's `.vo` object files

4.3 Other Use Cases

While it is common to view tools that perform pure computations (such as functional program execution or proof search a la λProlog) as producing assertions without proofs, there are various well-known reasoning systems that have been used a lot without being either certified or certifying: for example, Twelf [33]. DAMF would enable Twelf-based assertions to be exported to agents willing to trust its type and totality checkers.

The relationship of DAMF to the following topics is discussed in greater detail in the technical report [3]: libraries as curation on top of the DAMF model of global objects; attacks in the adversarial environment of the web; and possible uses of this framework in settings (such as journalism) where the lack of formal proof means increasing the need to explicitly track trust.

5 Related Work

The *semantic web* [14,15] was proposed to enrich the web with aspects of trust and would rely on concepts and technologies such as cryptography, taxonomies, ontologies, and inference rules. While the semantic web and DAMF both use cryptographic signatures and low-level web-based technologies, DAMF differs from the semantic web by focusing on objects rather than documents and using richer notions of logic and compositional reasoning.

Dedukti [8] is a dependently typed λ-calculus augmented with rewriting. Dedukti can be used to produce adapters (Sect. 2.3): in particular, proofs in a source system can be transformed to Dedukti proofs and then transformed back into formal proofs in a different system. For example, the Logipedia documentation mentions that "some proofs expressed in some Dedukti theories can be translated to other proof systems, such as HOL Light, HOL 4, Isabelle/HOL, Coq, Matita, Lean, PVS, ..." [29]. As a by-product, Dedukti can be used to build correctness-preserving translations of assertions for DAMF.

TPTP [40] provides a number of standards for the concrete syntax of first-order and higher-order logic along with tools for parsing and printing files that adhere to such standards. Deploying those tools for the production of the kind of multilingual adapters that we have described in Sect. 2.3 is a natural next step for tool development within DAMF.

The recognition that distributing some aspects of proof environments goes back to at least the systems described by Sacerdoti Coen, et al. [7,19]. In such systems, integration was meant to work between "near-peer" systems: that is, between systems that are both based on rich logics such as higher-order logic or on typed λ-calculi based on the Curry-Howard correspondence. A prerequisite for successful integration in such systems is the ability to connect the semantics of formulas, types, universes, proofs, etc. The wide spread use of such integration approaches has been delayed since it has only been in recent years that efforts, such as Dedukti [8] and MMT [36,37], are making it possible to form the necessary deep and sophisticated ties between the semantics of these objects arising from different implementations. In contrast, DAMF allows the

composition of different assertions without an a priori assumption that there is a formal semantics that relates them. Of course, correctness is a concern in many (most) situations: in those cases, Dedukti and MMT encodings can be used to translate assertions between two provers with precise correctness assurances. Often, however, the integration is of a more asymmetric kind. For example, when integrating a system that only performs integer operations or reasons only with integer inequalities (operations that are available in SMT systems) with a system based on higher-order logic, producing adapters based on sophisticated encodings might be completely unnecessary. The DAMF system similarly allows such integration.

6 Conclusion

We have described a Distributed Assertion Management Framework (DAMF) designed to share assertions between agents while tracking dependencies with canonical content ids (`cids`). This framework endows assertions with reliable provenance using public key cryptography and distributes them globally using the IPFS network. We have given an example of using DAMF to import a Coq lemma into Abella. The biggest challenge for future work is to adapt existing work on language translation and proof translation (in, e.g., Dedukti) to create or derive adapters automatically. Another important matter for future consideration is whether to persist compositions (i.e., Compose-derivations, cf. Sect. 2.4) to DAMF, which can serve as hints for post hoc investigations.

References

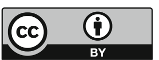
1. Abadi, M.: Variations in access control logic. In: van der Meyden, R., van der Torre, L. (eds.) DEON 2008. LNCS (LNAI), vol. 5076, pp. 96–109. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70525-3_9
2. Abel, A., et al.: POPLMark reloaded: mechanizing proofs by logical relations. J. Funct. Program. **29**, e19 (2019). https://doi.org/10.1017/S0956796819000170
3. Al Wardani, F., Chaudhuri, K., Miller, D.: Distributing and trusting proof checking: a preliminary report. Technical report, Inria Saclay (2022). https://hal.inria.fr/hal-03909741
4. Al Wardani, F., Chaudhuri, K., Miller, D.: Formal reasoning using distributed assertions. Technical report. HAL-04167922, Inria (2023). https://inria.hal.science/hal-04167922
5. ANSSI, F.N.C.A.: Requirements on the use of Coq in the context of common criteria evaluations. URL (2021). v1.1
6. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12
7. Asperti, A., Padovani, L., Coen, C.S., Guidi, F., Schena, I.: Mathematical knowledge management in HELM. Ann. Math. Artif. Intell. **38**(1–3), 27–46 (2003)
8. Assaf, A., et al.: Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory (2016). http://www.lsv.ens-cachan.fr/dowek/Publi/expressing.pdf

9. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_4
10. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: a system for reasoning about relational specifications. J. Formaliz. Reason. **7**(2), 1–89 (2014). https://doi.org/10.6092/issn.1972-5787/4650
11. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Trans. A R. Soc. **363**(1835), 2351–2375 (2005)
12. Barendregt, H., Barendsen, E.: Autarkic computations in formal proofs. J. Autom. Reason. **28**(3), 321–336 (2002). https://doi.org/10.1023/A:1015761529444
13. Benet, J.: IPFS-content addressed, versioned, P2P file system (2014). https://doi.org/10.48550/arxiv.1407.3561
14. Berners-Lee, T.: Semantic Web road map. Technical report, W3C Design Issues (1998). http://www.w3.org/DesignIssues/Semantic.html
15. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American Magazine (May 2001)
16. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA + proof system: building a heterogeneous verification platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, p. 44. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14808-8_3
17. Chaudhuri, K., Gérard, U., Miller, D.: Computation-as-deduction in Abella: work in progress. In: 13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. Oxford, United Kingdom, July 2018. https://hal.inria.fr/hal-01806154
18. Chihani, Z., Miller, D., Renaud, F.: A semantic framework for proof evidence. J. Autom. Reason. **59**(3), 287–330 (2016). https://doi.org/10.1007/s10817-016-9380-6
19. Coen, C.S.: Mathematical libraries as proof assistant environments. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 332–346. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27818-4_24
20. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 275–294. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_18
21. Dowek, G., Thiré, F.: Logipedia: a multi-system encyclopedia of formal proofs. http://www.lsv.fr/dowek/Publi/logipedia.pdf (2019)
22. Felty, A.P., Momigliano, A., Pientka, B.: The next 700 challenge problems for reasoning with higher-order abstract syntax representations. J. Autom. Reason. **55**(4), 307–372 (2015). https://doi.org/10.1007/s10817-015-9327-3
23. Felty, A.P., Momigliano, A., Pientka, B.: Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. Math. Struct. Comput. Sci. **28**, 1507–1540 (2017). https://doi.org/10.1017/S0960129517000093
24. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
25. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_11
26. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. Inf. Comput. **209**(1), 48–73 (2011). https://doi.org/10.1016/j.ic.2010.09.004

27. Harrison, J.: The HOL Light tutorial (2017). https://www.cl.cam.ac.uk/jrh13/hol-light/tutorial.pdf
28. 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
29. Logipedia in a nutshell (2022). http://logipedia.inria.fr/about/about.php
30. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Trans. Comput. Log. **6**(4), 749–783 (2005). https://doi.org/10.1145/1094622.1094628
31. Momigliano, A., Pientka, B., Thibodeau, D.: A case study in programming coinductive proofs: Howe's method. Math. Struct. Comput. Sci. **29**(8), 1309–1343 (2019). https://doi.org/10.1017/S0960129518000415
32. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the Combined KEAPPA - IWIL Workshops. CEUR Workshop Proceedings, vol. 418, pp. 123–132. CEUR-WS.org (2008). http://ceur-ws.org/Vol-418/paper10.pdf
33. 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
34. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **186**(2), 165–193 (2003)
35. Rabe, F.: The future of logic: foundation-independence. Log. Univers. **10**(1), 1–20 (2016)
36. Rabe, F.: How to identify, translate and combine logics? J. Log. Comput. **27**(6), 1753–1798 (2017)
37. Rabe, F.: The MMT Language and System (2022). https://uniformal.github.io/
38. Rushby, J.: An evidential tool bus. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 36–36. Springer, Heidelberg (2005). https://doi.org/10.1007/11576280_3
39. Sozeau, M., et al.: The MetaCoq Project. J. Autom. Reason. **64**(5), 947–999 (2020). https://doi.org/10.1007/s10817-019-09540-0
40. 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). https://doi.org/10.1007/s10817-009-9143-8
41. Tiu, A.: On the role of names in reasoning about λ-tree syntax specifications. In: Abel, A., Urban, C. (eds.) International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008), pp. 32–46 (2008)
42. The Twelf project (2016). http://twelf.org/

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.



An Abstract CNF-to-d-DNNF Compiler Based on Chronological CDCL

Sibylle Möhle(✉)

Max Planck Institute for Informatics, Saarland Informatics Campus E1 4, 66123 Saarbrücken, Germany
smoehle@mpi-inf.mpg.de

Abstract. We present Abstract CNF2dDNNF, a calculus describing an approach for compiling a formula in conjunctive normal form (CNF) into deterministic negation normal form (d-DNNF). It combines component-based reasoning with a model enumeration approach based on conflict-driven clause learning (CDCL) with chronological backtracking. Its properties, such as soundness and termination, carry over to implementations which can be modeled by it. We provide a correctness proof and a detailed example. The main conceptual differences to currently available tools targeting d-DNNF compilation are discussed and future research directions presented. The aim of this work is to lay the theoretical foundation for a novel method for d-DNNF compilation. To the best of our knowledge, our approach is the first knowledge compilation method using CDCL with chronological backtracking.

Keywords: Knowledge compilation · d-DNNF · Chronological CDCL

1 Introduction

In real-world applications, constraints may be modeled in conjunctive normal form (CNF), but many tasks relevant in AI and reasoning, such as checks for consistency, validity, clausal entailment, and implicants, can not be executed efficiently on them [9]. Tackling these and other computationally expensive problems is the aim of the knowledge compilation paradigm [13]. The idea is to translate a formula into a language in which the task of interest can be executed efficiently [22]. The knowledge compilation map [22] contains an in-depth discussion of such languages and their properties, and other (families of) languages have been introduced since its publication [21,25,29]. The focus in this work is on the language deterministic decomposable negation normal form (d-DNNF) [19]. It has been applied in planning [2,39], Bayesian reasoning [15], diagnosis [3,43], and machine learning [28] as well as in functional E-MAJSAT [40], to mention a few, and was also studied from a theoretical perspective [7,8,10]. Several d-DNNF compilers are available [20,30,37,48], as well as a d-DNNF reasoner[1].

[1] http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 195–213, 2023.
https://doi.org/10.1007/978-3-031-43369-6_11

Translating a formula from CNF to d-DNNF requires to process the search space exhaustively. The number of variable assignments which need to be checked is exponential in the number of variables occurring in the formula and testing them one by one is out of question from a computational complexity point of view. However, if the formula can be partitioned into subformulae defined over pairwise disjoint sets of variables, these subformulae can be processed independently and the results combined [4]. This may reduce the amount of work per computation significantly. Consider $F = (a \vee b) \wedge (c \vee d)$ defined over the set of variables $V = \{a, b, c, d\}$. Its search space consists of $2^4 = 16$ variable assignments. The formula F can be partitioned into $F_1 = (a \vee b)$ and $F_2 = (c \vee d)$ defined over the sets of variables $V_1 = \{a, b\}$ and $V_2 = \{c, d\}$, respectively, and such that $F = F_1 \wedge F_2$. Due to $V_1 \cap V_2 = \emptyset$, d-DNNF representations of F_1 and F_2 can be computed independently and conjoined obtaining a d-DNNF representation of F. Moreover, in each computation we only need to check $2^2 = 4$ assignments. The subformulae F_1 and F_2 are called *components* due to the original motivation originating in graph theory, and the partitioning process is referred to as *decomposition* or *component analysis*. This approach, also called *component-based reasoning*, is realized in various exact #SAT solvers [1,4,11,12,41,42,47], and its success suggests that formulae stemming from real-world applications decompose well enough to generate a substantial amount of work saving.

The formula F in our example satisfies *decomposability* [22], i.e., for each conjunction, the conjuncts are defined over pairwise disjoint sets of variables. We call such a formula *decomposable*. Negations occur only in front of literals, hence it is in decomposable negation normal form (DNNF) [17,18]. A formula in which for each disjunction its disjuncts are pairwise logically contradictory satisfies *determinism* [22], i.e., for each disjunction $C_1 \vee \ldots \vee C_n$ it holds that $C_i \wedge C_j \equiv \bot$ for $i, j \in \{1, \ldots, n\}$ and $i \neq j$. A deterministic DNNF formula is said to be in d-DNNF. Determinism is also met by the language disjoint sum of products (DSOP), which is a disjunction of pairwise contradictory conjunctions of literals, and which is relevant in circuit design [5]. In a previous work [34], we introduced an approach for translating a CNF formula into DSOP based on CDCL with chronological backtracking. The motivation for using chronological backtracking is twofold. First, it has shown not to significantly harm solver performance [33,38]. Second, pairwise disjoint models are detected without the need for blocking clauses commonly used in model enumeration based on CDCL with non-chronological backtracking. Blocking clauses rule out already found models, but they also slow down the solver, and avoiding their usage in model enumeration by means of CDCL with chronological backtracking has empirically shown to be effective [46]. Enhancing our former approach [34] by component-based reasoning enables us to compute a d-DNNF representation of a CNF formula. Reconsider our previous example, and suppose we obtained $\mathsf{dsop}(F_1) = a \vee (\neg a \wedge b)$ and $\mathsf{dsop}(F_2) = c \vee (\neg c \wedge d)$. Now $F \equiv F_1 \wedge F_2$, hence $F \equiv \mathsf{dsop}(F_1) \wedge \mathsf{dsop}(F_2) = (a \vee (\neg a \wedge b)) \wedge (c \vee (\neg c \wedge d))$, which is in d-DNNF.

Our Contributions. We present Abstract CNF2dDNNF, ACD for short, a declarative formal framework describing the compilation of CNF into d-DNNF

and a proof of its correctness. This abstract presentation allows for a thorough understanding of our method at a conceptual level and of its correctness. If our framework is sound, every implementation which can be modeled by it is sound as well. This comprises optimizations and implementation details, such as caches. ACD combines component-based reasoning and CNF-to-DSOP compilation based on conflict-driven clause learning (CDCL) with chronological backtracking. Disjunctions with pairwise contradictory disjuncts are introduced by decisions and subsequently flipping their value upon backtracking, while conjunctions whose conjuncts share no variable are introduced by unit propagation and decomposition. For the sake of simplicity, in our calculus formulae are partitioned into two subformulae. However, lifting it to an arbitrary number of subcomponents is straightforward, and a corresponding generalization is presented.

2 Preliminaries

Let V be a set of propositional variables defined over the set of Boolean constants $\bot$ (false) and $\top$ (true) denoted by $\mathbb{B} = \{\bot, \top\}$. A *literal* is either a variable $v \in V$ or its negation $\neg v$. We refer to the variable of a literal ℓ by $\mathsf{var}(\ell)$ and extend this notation to sets and sequences of literals and formulae. We consider formulae in *conjunctive normal form (CNF)* which are conjunctions of *clauses* which are disjunctions of literals. A formula in *disjoint sum of products (DSOP)* is a disjunction of pairwise contradictory *cubes*, which are conjunctions of literals. Our target language is *deterministic decomposable negation normal form (d-DNNF)*, whose formulae are built of literals, conjunctions sharing no variables, and disjunctions whose disjuncts are pairwise contradictory. We might interpret formulae as sets of clauses and cubes and clauses and cubes as sets of literals by writing $C \in F$ and $\ell \in C$ to refer to a clause C in a formula F and a literal ℓ contained in a clause or cube C, respectively. The empty CNF formula and the empty cube are denoted by $\top$ and the empty DSOP formula and the empty clause by $\bot$.

A *total variable assignment* is a mapping $\sigma \colon V \mapsto \mathbb{B}$, and a *trail* $I = \ell_1 \dots \ell_n$ is a non-contradictory sequence of literals which might also be interpreted as a (possibly partial) assignment, such that $I(\ell) = \top$ iff $\ell \in I$. Similarly, $I(C)$ and $I(F)$ are defined. We might interpret a trail I as a set of literals and write $\ell \in I$ to refer to the literal ℓ on I. The empty trail is denoted by ε and the set of variables of the literals on I by $\mathsf{var}(I)$. Trails and literals can be concatenated, written $I\,J$ and $I\,\ell$, given $\mathsf{var}(I) \cap \mathsf{var}(J) = \emptyset$ and $\mathsf{var}(I) \cap \mathsf{var}(\ell) = \emptyset$. The position of ℓ on the trail I is denoted by $\tau(I, \ell)$. The decision literals on I are annotated by a superscript, e.g., ℓ^d, denoting open "left" branches in the sense of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [23,24]. *Flipping the value of a decision literal* can be seen as closing the corresponding left branch and starting a "right" branch, where the decision literal ℓ^d becomes a *flipped literal* $\neg\ell$.

The *residual* of F under I, written $F|_I$, is obtained by assigning the variables in F their truth value and by propagating truth values through Boolean

connectives. The notion of residual is extended to clauses and literals. A *unit clause* is a clause $\{\ell\}$ containing one single literal ℓ. By $\mathsf{units}(F)$ ($\mathsf{units}(F|_I)$) we denote the set of unit literals in F ($F|_I$). Similarly, $\mathsf{decs}(I)$ denotes the set of decision literals on I. By writing $\ell \in \mathsf{decs}(I)$ ($\ell \in \mathsf{units}(F)$, $\ell \in \mathsf{units}(F|_I)$), we refer to a decision literal ℓ on I (unit literal in F, $F|_I$). A trail I *falsifies* F, if $I(F) \equiv \bot$, i.e., $F|_I = \bot$. It *satisfies* F, $I \models F$, if $I(F) \equiv \top$, i.e., $F|_I = \top$, and is then called a *model* of F. If $\mathsf{var}(I) = V$, I is a *total model*, otherwise it is a *partial model*.

The trail is partitioned into *decision levels*, starting with a decision literal and extending until the literal preceding the next decision. The *decision level function* $\delta \colon V \mapsto \mathbb{N} \cup \{\infty\}$ returns the decision level of a variable $v \in V$. If v is unassigned, $\delta(v) = \infty$, and δ is updated whenever a variable is assigned or unassigned, e.g., $\delta[v \mapsto d]$ if v is assigned to decision level d. We define $\delta(\ell) = \delta(\mathsf{var}(\ell))$, $\delta(C) = \mathsf{max}\{\delta(\ell) \mid \ell \in C\}$ for $C \neq \bot$ and $\delta(I) = \mathsf{max}\{\delta(\ell) \mid \ell \in I\}$ for $I \neq \varepsilon$ extending this notation to sets of literals. Finally, we define $\delta(\bot) = \delta(\varepsilon) = \infty$. By writing $\delta[I \mapsto \infty]$, all literals on the trail I are unassigned. The decision level function is left-associative, i.e., $\delta[I \mapsto \infty][\ell \mapsto d]$ expresses that first all literals on I are unassigned and then literal ℓ is assigned to decision level d.

Unlike in CDCL with non-chronological backtracking [36,44,45], in *chronological CDCL* [33,38] literals may not be ordered on the trail in ascending order with respect to their decision level. We write $I_{\leqslant n}$ ($I_{<n}$, $I_{=n}$) for the subsequence of I containing all literals ℓ with $\delta(\ell) \leq n$ ($\delta(\ell) < n$, $\delta(\ell) = n$). The *pending search space* of I is given by the assignments not yet tested [34], i.e., I and its open right branches $R(I)$, and is defined as $O(I) = I \vee R(I)$, where $R(I) = \bigvee_{\ell \in \mathsf{decs}(I)} R_{=\delta(\ell)}(I)$ and $R_{=\delta(\ell)}(I) = \neg\ell \wedge I_{<\delta(\ell)}$ for $\ell \in \mathsf{decs}(I)$. As an example, for $I = a\, b^d\, c\, d\, e^d f$, $O(I) = (a \wedge b \wedge c \wedge d \wedge e \wedge f) \vee (\neg b \wedge a) \vee (\neg e \wedge a \wedge b \wedge c \wedge d)$. Similarly, the *pending models* of F are the satisfying assignments of F not yet detected and which are given by $F \wedge O(I)$.

3 Chronological CDCL for CNF-to-d-DNNF Compilation

In *static component analysis* the component structure is computed once, typically as a preprocessing step, and not altered during the further execution. In contrast, in our approach the component structure is computed iteratively adopting *dynamic component analysis*. Algorithm 1 provides a general schema in pseudo-code. It is formulated recursively, capturing the recursive nature of dynamic component analysis. Lines 1–7 and 11 describe model enumeration based on chronological CDCL [34], while lines 8–10 capture component analysis.

Now assume unit propagation has been carried out until completion, no conflict has occurred and there are still unassigned variables (line 8). If $F|_I$ can be decomposed into two formulae G and H, we call $\mathsf{CNF2dDNNF}$ recursively on G and H, conjoin the outcomes of these computations with I and add the result to M (line 9). If I contains no decisions, the search space has been explored exhaustively, otherwise chronological backtracking occurs (lines 10). The working of our approach is shown by an example.

Algorithm 1: CNF2dDNNF(F, V, I, M)

---

input : CNF $F, V = \mathsf{var}(F), I = \varepsilon, M = \bot$
output: d-DNNF $M \equiv F$

1 **Loop**
2   $I \leftarrow$ PropagateUnits()
3   **if** *conflict occurs* **then**
4   **if** *conflict level* $= 0$ **then return** M **else** AnalyzeConflict()
5   **else if** $I(F) = \top$ **then**
6     $M \leftarrow M \vee I$
7   **if** *there are no decisions on* I **then return** M **else** BacktrackChrono()
8   **else if** $F|_I$ *can be decomposed into* G *and* H **then**
9     $M \leftarrow M \vee I \wedge$ CNF2dDNNF$(G, \mathit{var}(G), \varepsilon, \bot) \wedge$ CNF2dDNNF$(H, \mathit{var}(H), \varepsilon, \bot)$
10   **if** *there are no decisions on* I **then return** M **else** BacktrackChrono()
11   **else** Decide

Example 1. Let $V = \{a, b, c, d, e, f, g, h\}$ be a set of propositional variables and $F = (a) \wedge (\neg a \vee \neg b \vee c \vee d) \wedge (\neg a \vee \neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f) \wedge (g \vee h)$ be a formula defined over V. The execution is depicted as a tree in Fig. 1. For the sake of readability, we show only the formula on which a rule is executed, represented by a box annotated with its component level. Black arrows correspond to "downward" rule applications, while violet (gray) arrows represent "upwards" rule applications and are annotated with the formula returned by the computation of a component. Ignore the rule names for now, they are intended to clarify the working of our calculus which is presented in Sect. 4. We see that, first, a is propagated, denoted by the black vertical arrow annotated with a and the name of the applied rule (Unit). The residual of F under a is $F|_a = (\neg b \vee c \vee d) \wedge (\neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f) \wedge (g \vee h)$ (not shown). It contains no unit clause but can be decomposed into $(\neg b \vee c \vee d) \wedge (\neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f)$ and $(g \vee h)$. Two new (sub)components are created (by applying rule Decompose) with component level 01 and 02, respectively, represented by the shadowed boxes.

Since $(g \vee h)$ can not be decomposed further, model enumeration with chronological CDCL is executed on it (not shown) by deciding g (rule Decide) satisfying $(g \vee h)$, followed by backtracking chronologically (BackTrue), which amounts to negating the value of the most recent decision g, and propagating h (Unit). The processing of $(g \vee h)$ terminates with $g \vee \neg g \wedge h$ (CompTrue, not shown). But before this result can be used further, the subcomponent at component level 01 needs to be processed. Its formula is $G = (\neg b \vee c \vee d) \wedge (\neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f)$. It neither contains a unit nor can it be decomposed, hence we take a decision, let's say, b^d. Now $G|_b = (c \vee d) \wedge (e \vee f)$, which is decomposed into two components with one clause each and component level 011 and 012, respectively (Decompose). These formulae can not be decomposed further, and they are processed independently, similarly to $(g \vee h)$. Before G was decomposed, a decision was taken, and we backtrack combining the results of its subcomponents (ComposeBack). We have $G|_{\neg b} = (\neg c \vee e) \wedge (d \vee f)$ resulting in two components with component

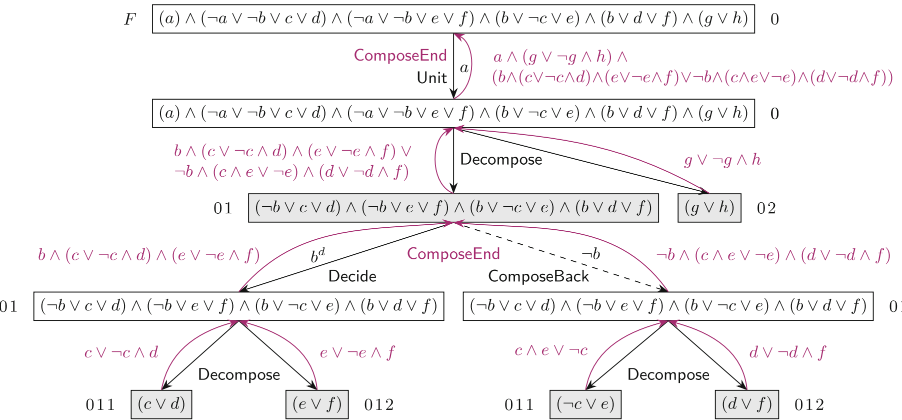


Fig. 1. Component structure of F created by ACD.

levels 011 and 012, respectively. They are processed and their results combined, after which the results of the subcomponents of the root component are conjoined with a. There is no decision on the trail, and the process terminates with $M = (a) \wedge (\neg a \vee \neg b \vee c \vee d) \wedge (\neg a \vee \neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f) \wedge (g \vee h)$ (ComposeEnd). Notice that although component levels can occur multiple times throughout the computation, they are unique at any point in time.

4 Calculus

Due to its recursive nature, combining the results computed for subcomponents in CNF2dDNNF is straightforward. For its formalization, however, a non-recursive approach turned out to be better suited. Consequently, a method is needed for matching subcomponents and their parent. For this purpose, a *component level* is associated with each component. It is defined as a string of numbers in $\mathbb{N}$ as follows. Suppose a component $\mathcal{C}$ is assigned level "d" and assume its formula is decomposed into two subformulae. The corresponding subcomponents $\mathcal{C}_G$ and $\mathcal{C}_H$ are assigned component levels "$d \cdot 1$" and "$d \cdot 2$", respectively, with "$\cdot$" denoting string composition. Accordingly, the component level of their parent $\mathcal{C}$ is given by the substring consisting of all but the last element of their level, i.e., "d".[2] The *root component* holds the input formula, it has no parent and its component level is zero. A component is *closed* if no rule can be applied to it, and *decomposed* if either at least one of its subcomponents is not closed or both its subcomponents are closed, but their results are not yet combined. Components which are neither closed nor decomposed are *open*.[3] Closed components may be discarded as soon

[2] From now on, we omit the quotes for the sake of readability.

[3] The differentiation between open and decomposed components is purely technical and needed for the termination proof in Sect. 5.

as their results are combined, and the computation stops as soon as the root component is closed. With these remarks, we are ready to present our calculus.

We describe our algorithm in terms of a state transition system ABSTRACT CNF2dDNNF, ACD for short, over a set of global states S, a transition relation $\rightsquigarrow \subseteq S \times S$ and an initial global state $\mathcal{S}_0$. A *global state* is a set of components. A *component* $\mathcal{C}$ is described as a seven-tuple $(F, V, d, e, I, M, \delta)^s$, where s denotes its *component state*. It is c if $\mathcal{C}$ is closed, f if F is decomposed, and o if $\mathcal{C}$ is open. The first two elements F and V refer to a formula and its set of variables, respectively. The third element d denotes the component level of $\mathcal{C}$. If $d \neq 0$, then $d \in \{d' \cdot 1, d' \cdot 2\}$, where d' is the component level of the parent component of $\mathcal{C}$, as explained above. In this manner, the component level keeps track of the decomposition structure of F and is used to match parent components and their subcomponents. The number of subcomponents of $\mathcal{C}$ is given by e, while I and δ refer to a trail ranging over variables in V and a decision level function with domain V, respectively. Finally, M is a formula in d-DNNF representing the models of F found so far. A component is initialized by $(F, V, d, 0, \varepsilon, \bot, \infty)^o$ and closed after its computation has terminated, i.e., $(F, V, d, 0, I, M, \delta)^c$. Notice that in these cases $e = 0$. The *initial global state* $\mathcal{S}_0 = \{\mathcal{C}_0\}$ consists of the *root component* $\mathcal{C}_0 = (F, V, 0, 0, \varepsilon, \bot, \infty)^o$ with F and V denoting the input formula and $V = \mathsf{var}(F)$, while the *final global state* is given by $\mathcal{S}_n = \{(F, V, 0, 0, I, M, \delta)^c\}$ where $M \equiv F$ is in d-DNNF. The transition relation $\rightsquigarrow$ is defined as the union of transition relations $\rightsquigarrow_{\mathsf{R}}$, where R is either Unit, Decide, BackTrue, BackFalse, CompTrue, CompFalse, Decompose, ComposeBack or ComposeEnd. Our calculus contains three types of rules, which can abstractly be described as follows:

$$\alpha\colon \mathcal{S} \uplus \{\mathcal{C}\} \rightsquigarrow_{\mathsf{R}} \mathcal{S} \uplus \mathcal{C}';\ \beta\colon \mathcal{S} \uplus \{\mathcal{C}\} \rightsquigarrow_{\mathsf{R}} \mathcal{S} \uplus \{\mathcal{C}', \mathcal{C}_1, \mathcal{C}_2\};\ \gamma\colon \mathcal{S} \uplus \{\mathcal{C}, \mathcal{C}_1, \mathcal{C}_2\} \rightsquigarrow_{\mathsf{R}} \mathcal{S} \uplus \{\mathcal{C}'\}.$$

In this description, $\mathcal{S}$ refers to the subset of the current global state consisting of all components which are not touched by rule R, with $\uplus$ denoting the disjoint set union, e.g., in α, $\mathcal{C}, \mathcal{C}' \notin \mathcal{S}$. An α rule affects a component $\mathcal{C}$ turning it into $\mathcal{C}'$. The rules Unit, Decide, BackTrue, BackFalse, CompTrue, and CompFalse are α rules. A β rule modifies $\mathcal{C}$ obtaining $\mathcal{C}'$ and creates two new components $\mathcal{C}_1$ and $\mathcal{C}_2$. Rule Decompose is the only β rule. Finally, a γ rule removes the two components $\mathcal{C}_1$ and $\mathcal{C}_2$ from the global state and modifies their parent $\mathcal{C}$. Rules ComposeBack and ComposeEnd are γ rules. The rules are listed in Fig. 2.

Model Computation. Rules Unit, Decide, BackTrue, BackFalse, CompTrue, and CompFalse execute model enumeration with chronological CDCL [34] and are applicable exclusively to open components. Unit literals are assigned the decision level of their reason, which might be lower than the current decision level (rule Unit). Decisions can be taken only if the processed formula is not decomposable (Decide). Backtracking occurs chronologically, i.e., to the second highest decision level on the trail, after finding a model (BackTrue) and to the decision level preceding the conflict level after conflict analysis (BackFalse), respectively. In the latter case, the propagated literal is assigned the lowest level at which the learned clause becomes unit and to which a SAT solver implementing CDCL with

Unit: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{Unit}} \mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I\,\ell,\, M,\, \delta[\ell \mapsto a])^o\}$ if $\bot \notin F|_I$ and exists $C \in F$ with $\{\ell\} = C|_I$ and $a \overset{\text{def}}{=} \delta(C \setminus \{\ell\})$ if $C \setminus \{\ell\} \neq \bot$ and $a \overset{\text{def}}{=} 0$ otherwise

Decide: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{Decide}} \mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I\,\ell^d,\, M,\, \delta[\ell \mapsto a])^o\}$ if $F|_I \neq \top$ and $\bot \notin F|_I$ and $\mathsf{units}(F|_I) = \emptyset$ and $\mathsf{var}(\ell) \in V$ and $\delta(\ell) = \infty$ and $a \overset{\text{def}}{=} \delta(I) + 1$ and there exist no G and H such that $G \wedge H = F|_I$ and $\mathsf{var}(G) \cap \mathsf{var}(H) = \emptyset$

BackTrue: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{BackTrue}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, P\,K\,\ell,\, M \vee I,\, \delta[L \mapsto \infty][\ell \mapsto e])^o\}$ if $F|_I = \top$ and $PQ \overset{\text{def}}{=} I$ and $D \overset{\text{def}}{=} \neg\mathsf{decs}(I)$ and $e + 1 \overset{\text{def}}{=} \delta(D) = \delta(I)$ and $\ell \in D$ and $e = \delta(D \setminus \{\ell\}) = \delta(P)$ and $K \overset{\text{def}}{=} Q_{\leqslant e}$ and $L \overset{\text{def}}{=} Q_{>e}$

BackFalse: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{BackFalse}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, P\,K\,\ell,\, M,\, \delta[L \mapsto \infty][\ell \mapsto j])^o\}$ if exists $C \in F$ and exists D with $PQ \overset{\text{def}}{=} I$ and $C|_I = \bot$ and $c \overset{\text{def}}{=} \delta(C) = \delta(D) > 0$ such that $\ell \in D$ and $\neg\ell \in \mathsf{decs}(I)$ and $\ell|_Q = \bot$ and $F \wedge \neg M \models D$ and $j \overset{\text{def}}{=} \delta(D \setminus \{\ell\})$ and $b \overset{\text{def}}{=} \delta(P) = c - 1$ and $K \overset{\text{def}}{=} Q_{\leqslant b}$ and $L \overset{\text{def}}{=} Q_{>b}$

CompTrue: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{CompTrue}} \mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M \vee I,\, \delta)^c\}$ if $F|_I = \top$ and $\mathsf{decs}(I) = \emptyset$

CompFalse: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{CompFalse}} \mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^c\}$ if exists $C \in F$ and $C|_I = \bot$ and $\delta(C) = 0$

Decompose: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \rightsquigarrow_{\mathsf{Decompose}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 2,\, I,\, M,\, \delta)^f, (G,\, U,\, d{\cdot}1,\, 0,\, \varepsilon,\, \bot,\, \infty)^o, (H,\, W,\, d{\cdot}2,\, 0,\, \varepsilon,\, \bot,\, \infty)^o\}$ if $F|_I \neq \top$ and $\bot \notin F|_I$ and $\mathsf{units}(F|_I) = \emptyset$ and $G \wedge H \overset{\text{def}}{=} F|_I$ and $U \overset{\text{def}}{=} \mathsf{var}(G)$ and $W \overset{\text{def}}{=} \mathsf{var}(H)$ and $U \cap W = \emptyset$

ComposeBack: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 2,\, I,\, M,\, \delta_F)^f,$ $(G,\, U,\, d \cdot 1,\, 0,\, J_G,\, N,\, \delta_G)^c, (H,\, W,\, d \cdot 2,\, 0,\, J_H,\, O,\, \delta_H)^c\} \rightsquigarrow_{\mathsf{ComposeBack}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, P\,K\,\ell,\, M \vee (I \wedge N \wedge O),\, \delta[L \mapsto \infty][\ell \mapsto e])^o\}$ if $PQ \overset{\text{def}}{=} I$ and $D \overset{\text{def}}{=} \neg\mathsf{decs}(I)$ and $e + 1 \overset{\text{def}}{=} \delta(D) = \delta(I)$ and $\ell \in D$ and $e = \delta(D \setminus \{\ell\}) = \delta(P)$ and $K \overset{\text{def}}{=} Q_{\leqslant e}$ and $L \overset{\text{def}}{=} Q_{>e}$

ComposeEnd: $\mathcal{S} \uplus \{(F,\, V,\, d,\, 2,\, I,\, M,\, \delta)^f,$ $(G,\, U,\, d \cdot 1,\, 0,\, J_G,\, N,\, \delta_G)^c, (H,\, W,\, d \cdot 2,\, 0,\, J_H,\, O,\, \delta_H)^c\} \rightsquigarrow_{\mathsf{ComposeEnd}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M \vee (I \wedge N \wedge O),\, \delta)^c\}$ if $\mathsf{decs}(I) = \emptyset$

Fig. 2. ACD transition rules.

non-chronological backtracking would backtrack to. Since the literals might not be ordered on the trail in ascending order with respect to their decision level, a non-contiguous part of it is discarded. Finally, a component is closed if its trail contains no decisions and either satisfies its formula (CompTrue) or a conflict occurs at decision level zero, i.e., the conflicting clause has decision level zero (CompFalse). In the former case, the newly found model is recorded.

Component Analysis. Rules Decompose, ComposeBack, and ComposeEnd capture the decomposition of a formula and the combination of the models of its subformulae and thus affect multiple components.

Decompose. The state of the parent component $\mathcal{C}$ with formula F is o (open). The trail I neither satisfies nor falsifies F, and $F|_I$ contains no unit clause but can

be partitioned into two formulae G and H defined over disjoint sets of variables. Subcomponents for G and H are created, the number of subcomponents of $\mathcal{C}$ is set to two and its state is changed to f (decomposed). Notice that $\mathcal{C}$ can only be processed further after its subcomponents are closed.

ComposeBack. The state of the component $\mathcal{C}$ with formula F is f (decomposed). Its subcomponents $\mathcal{C}_G$ and $\mathcal{C}_H$ with formulae G and H, respectively, have state c (closed). Furthermore, $N \equiv G$ and $O \equiv H$, hence $F|_I \equiv I \wedge N \wedge O$, which is added to M. This corresponds to enumerating multiple models of F in one step. This can easily be seen by applying the distributive laws to $I \wedge N \wedge O$ which gives us a DSOP formula whose disjuncts are satisfying assignments of $F|_I$. The search space has not yet been processed exhaustively ($\delta(I) > 0$), backtracking to the second highest decision level occurs, and the state of $\mathcal{C}$ is changed back to o (open). Finally, $\mathcal{C}_G$ and $\mathcal{C}_H$ are removed from the global state. If I can not be extended to a model of F, we have $N = \bot$ or $O = \bot$, and $I \wedge N \wedge O = \bot$. Otherwise, $I \wedge N \wedge O \neq \bot$. Both cases are captured by rule ComposeBack.

ComposeEnd. The state of the parent component $\mathcal{C}$ with formula F is f (decomposed). Its subcomponents $\mathcal{C}_G$ and $\mathcal{C}_H$ with formulae G and H, respectively, are closed. Furthermore, $N \equiv G$ and $O \equiv H$, hence $F|_I \equiv I \wedge N \wedge O$, which is added to M. The search space has been processed exhaustively ($\mathsf{decs}(I) = \emptyset$), and the state of $\mathcal{C}$ is set to c (closed). Finally, $\mathcal{C}_G$ and $\mathcal{C}_H$ are removed from the global state. As in rule ComposeBack, either $I \wedge N \wedge O = \bot$ or $I \wedge N \wedge O \neq \bot$.

Example 2. Reconsider Example 1 with variables $V = \{a, b, c, d, e, f, g, h\}$ and $F = (a) \wedge (\neg a \vee \neg b \vee c \vee d) \wedge (\neg a \vee \neg b \vee e \vee f) \wedge (b \vee \neg c \vee e) \wedge (b \vee d \vee f) \wedge (g \vee h)$ defined over V. The execution trace of ACD is shown in Fig. 3. Unaffected components are depicted in gray, and model enumeration by means of chronological CDCL is shown only once in full detail. The execution starts with the root component $\mathcal{C}_F$ containing F. In step (1), the unit literal a is propagated, upon which $F|_a$ is decomposed into $(g \vee h)$ and G creating components $\mathcal{C}_{(g \vee h)}$ and $\mathcal{C}_G$ shown in (2). Steps (3) to (6) capture model enumeration by chronological CDCL of $(g \vee h)$, i.e., the computation of a DSOP representation of $(g \vee h)$, after which $\mathcal{C}_{(g \vee h)}$ is closed. Next, the formula G is processed by deciding b in step (7), decomposing $G|_b$ into $(c \vee d)$ and $(e \vee f)$ and creating components $\mathcal{C}_{(c \vee d)}$ and $\mathcal{C}_{(e \vee f)}$, respectively, in step (8). The processing of $\mathcal{C}_{(c \vee d)}$ and $\mathcal{C}_{(e \vee f)}$ occurs analogously to steps (3) to (6) resulting in the state shown in (9). The results are conjoined with b, which is the trail of $\mathcal{C}_G$ and under which $G|_b$ was decomposed. Since b is a decision, it is flipped in (10) to explore its right branch $\neg b$. The formula $G|_{\neg b}$ is decomposed into $(\neg c \vee e)$ and $(d \vee f)$ and components $\mathcal{C}_{(\neg c \vee e)}$ and $\mathcal{C}_{(d \vee f)}$ are created, as in (11). Their processing, which is not shown, results in the state depicted in (12), and the results are conjoined with the trail of $\mathcal{C}_G$. Since its trail contains no decision, $\mathcal{C}_G$ is closed, see (13). The global state now contains the root component and its two subcomponents, which are closed, hence the rule ComposeEnd is executed, and the computation terminates with the closed root component and $M = a \wedge (g \vee \neg g \wedge h) \wedge (b \wedge (c \vee \neg c \wedge d) \wedge (e \vee \neg e \wedge f) \vee \neg b \wedge (c \wedge e \vee \neg c) \wedge (d \vee \neg d \wedge f)$, where $M \equiv F$, and which is shown in (14).

$$\{(F,\,V,\,0,\,0,\,\varepsilon,\,\bot,\,\delta_{00}=\infty)^o\}$$

$$\rightsquigarrow_{\text{Unit}} \quad \{(F,\,V,\,0,\,0,\,a,\,\bot,\,\delta_{01}=\delta_{00}[a\mapsto 0])^o\} \tag{1}$$

$$\rightsquigarrow_{\text{Decompose}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\varepsilon,\,\bot,\,\delta_{020}=\infty)^o,\\ (G=(\neg b\vee c\vee d)\wedge(\neg b\vee e\vee f)\wedge(b\vee\neg c\vee e)\wedge(b\vee d\vee f),\,\{b,c,d,e,f\},\,01,\,0,\,\varepsilon,\,\bot,\,\delta_{010}=\infty)^o\} \tag{2}$$

// enumerate models of $(g\vee h)$ with chronological CDCL

$$\rightsquigarrow_{\text{Decide}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,g^d,\,\bot,\,\delta_{021}=\delta_{020}[g\mapsto 1])^o,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,\varepsilon,\,\bot,\,\delta_{010})^o\} \tag{3}$$

$$\rightsquigarrow_{\text{BackTrue}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g,\,g,\,\delta_{022}=\delta_{021}[g\mapsto\infty][g\mapsto 0])^o,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,\varepsilon,\,\bot,\,\delta_{010})^o\} \tag{4}$$

$$\rightsquigarrow_{\text{Unit}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g,\,\delta_{023}=\delta_{022}[h\mapsto 0])^o,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,\varepsilon,\,\bot,\,\delta_{010})^o\} \tag{5}$$

$$\rightsquigarrow_{\text{CompTrue}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,\varepsilon,\,\bot,\,\delta_{010})^o\} \tag{6}$$

$$\rightsquigarrow_{\text{Decide}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,b^d,\,\bot,\,\delta_{011}=\delta_{010}[b\mapsto 1])^o\} \tag{7}$$

$$\rightsquigarrow_{\text{Decompose}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,2,\,b^d,\,\bot,\,\delta_{011})^f,\\ ((c\vee d),\,\{c,d\},\,011,\,0,\,\varepsilon,\,\bot,\,\delta_{0110}=\infty)^o,\,((e\vee f),\,\{e,f\},\,012,\,0,\,\varepsilon,\,\bot,\,\delta_{0120}=\infty)^o\} \tag{8}$$

// enumerate models of $(c\vee d)$ and $(e\vee f)$ with chronological CDCL

...

$$\rightsquigarrow_{\text{CompTrue}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,2,\,b^d,\,\bot,\,\delta_{011})^f,\\ ((c\vee d),\,\{c,d\},\,011,\,0,\,\neg c d,\,c\vee\neg c\wedge d,\,\delta_{0113})^c,\,((e\vee f),\,\{e,f\},\,012,\,0,\,\neg e f,\,e\vee\neg e\wedge f,\,\delta_{0123})^c\} \tag{9}$$

// combine results

$$\rightsquigarrow_{\text{ComposeBack}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,0,\,\neg b,\,b\wedge(c\vee\neg c\wedge d)\wedge(e\vee\neg e\wedge f),\,\delta_{012}=\delta_{011}[b\mapsto\infty][b\mapsto 0])^o\} \tag{10}$$

$$\rightsquigarrow_{\text{Decompose}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,2,\,\neg b,\,b\wedge(c\vee\neg c\wedge d)\wedge(e\vee\neg e\wedge f),\,\delta_{012})^f,\\ ((\neg c\vee e),\,\{c,e\},\,011,\,0,\,\varepsilon,\,\bot,\,\delta_{0110}=\infty)^o,((d\vee f),\,\{d,f\},\,012,\,0,\,\varepsilon,\,\bot,\,\delta_{0120}=\infty)^o \tag{11}$$

// enumerate models of $(\neg c\vee e)$ and $(d\vee f)$ with chronological CDCL

...

$$\rightsquigarrow_{\text{CompTrue}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,2,\,\neg b,\,b\wedge(c\vee\neg c\wedge d)\wedge(e\vee\neg e\wedge f),\,\delta_{012})^f,\\ ((\neg c\vee e),\,\{c,e\},\,011,\,0,\,\neg c,\,c\wedge e\vee\neg c,\,\delta_{0113})^c,\,((d\vee f),\,\{d,f\},\,012,\,0,\,\neg d f,\,d\vee\neg d\wedge f,\,\delta_{0123})^c\} \tag{12}$$

// combine results

$$\rightsquigarrow_{\text{ComposeEnd}} \quad \{(F,\,V,\,0,\,2,\,a,\,\bot,\,\delta_{01})^f,\,((g\vee h),\,\{g,h\},\,02,\,0,\,\neg g h,\,g\vee\neg g\wedge h,\,\delta_{023})^c,\\ (G,\,\{b,c,d,e,f\},\,01,\,2,\,\neg b,\,b\wedge(c\vee\neg c\wedge d)\wedge(e\vee\neg e\wedge f)\vee\neg b\wedge(c\wedge e\vee\neg c)\wedge(d\vee\neg d\wedge f),\,\delta_{012})^c\} \tag{13}$$

$$\rightsquigarrow_{\text{ComposeEnd}} \quad \{(F,\,V,\,0,\,2,\,a,\,a\wedge(g\vee\neg g\wedge h)\wedge(b\wedge(c\vee\neg c\wedge d)\wedge(e\vee\neg e\wedge f)\vee\neg b\wedge(c\wedge e\vee\neg c)\wedge(d\vee\neg d\wedge f),\,\delta_{012})^c\} \tag{14}$$

Fig. 3. Execution trace of ACD for Example 1.

5 Proofs

For proving correctness, we first show that our calculus is sound by identifying invariants which need to hold in a sound global state and show that they still hold after the execution of any rule. Then we prove that for any closed component it holds that $M\equiv F$ and that ACD can not get stuck and terminates in a correct state. Showing termination concludes our proof.

Definition 1 (Sound Global State). *A global state $\mathcal{S}$ is* sound *if for all its components $\mathcal{C}=(F,\,V,\,d,\,e,\,I,\,M,\,\delta)^s$ the following invariants hold:*

(1) $\forall k,\ell\in \mathsf{decs}(I)\;.\;\tau(I,k)<\tau(I,\ell)\implies\delta(k)<\delta(\ell)$
(2) $\delta(\mathsf{decs}(I))=\{1,\ldots,\delta(I)\}$
(3) $\forall n\in\mathbb{N}\;.\;F\wedge\neg M\wedge\mathsf{decs}_{\leqslant n}(I)\models I_{\leqslant n}$*, provided $\mathcal{C}$ is open or decomposed*

(4) $M \vee O(I)$ *is a d-DNNF, provided* $\mathcal{C}$ *is open or decomposed*
(5) $M \vee F \wedge O(I) \equiv F$
(6) $e > 0$ *iff (A)* $e = 2$*, (B)* $\mathcal{C}$ *is decomposed, (C)* $\mathcal{S}$ *contains components* $\mathcal{C}_G = (G,\, \mathsf{var}(G),\, d \cdot 1,\, e_g,\, J_G,\, N,\, \delta_G)^s$*,* $\mathcal{C}_H = (H,\, \mathsf{var}(H),\, d \cdot 2,\, e_H,\, J_H,\, O,\, \delta_H)^s$*, such that* $F|_I = G \wedge H$ *and* $\mathsf{var}(G) \cap \mathsf{var}(H) = \emptyset$
(7) If $e = 2$ *and* $\mathcal{S}$ *contains components* $\mathcal{C}_G = (G,\, \mathsf{var}(G),\, d \cdot 1,\, 0,\, J_G,\, N,\, \delta_G)^c$ *and* $\mathcal{C}_H = (H,\, \mathsf{var}(H),\, d \cdot 2,\, 0,\, J_H,\, O,\, \delta_H)^c$*, then* $F|_I \equiv I \wedge N \wedge O$
(8) If $\mathcal{C}$ *is closed, then* $\mathsf{decs}(I) = \emptyset$

Invariants (1) - (5) correspond to the ones in our previous work [34]. They say that decisions are ordered in ascending order with respect to their decision level and that every decision level contains a decision literal. They further ensure that literals propagated after backtracking upon finding a model are indeed implied, that no model is enumerated multiple times and that all models are found. Invariant (3) is only useful for open or decomposed components, since I remains unaltered when a component is closed. Invariant (4) only holds for closed components if $I(F) = \bot$. Invariants (6) and (7) are concerned with the properties of a parent component and its subcomponents (for the case $c = 2$), such as the definition of the component level. Since, given a trail I, $F|_I$ is decomposed into formulae G and H, we also have that $F|_I \equiv N \wedge O$, where $N \equiv G$ and $O \equiv H$. Finally, Inv. (8) says that the trail of a closed component contains no decision.

Lemma 1 (Soundness of the Initial Global State). *The initial global state* $\mathcal{S}_0 = \{(F,\, V,\, 0,\, 0,\, \varepsilon,\, \bot,\, \infty)^o\}$ *is sound.*

Proof. Due to $I = \varepsilon$ and $e = 0$ and since the (root) component is open, all invariants in Definition 1 are trivially met.

Theorem 1 (Soundness of ACD Rules). *The rules of* ACD *preserve soundness, i.e., they transform a sound global state into another sound global state.*

Proof. The proof is carried out by induction over the rule applications. We assume that prior to the application of a rule the invariants in Definition 1 are met and show that they also hold in the target state. The (parent) component in the original state is denoted by $\mathcal{C} = (F,\, V,\, d,\, e,\, I,\, M,\, \delta)^s$ and in the target state by $\mathcal{C}' = (F,\, V,\, d',\, e',\, I',\, M',\, \delta')^{s'}$. Its subcomponents, if there are any, are written $\mathcal{C}_G = (G,\, \mathsf{var}(G),\, d \cdot 1,\, e_G,\, J,\, N,\, \delta_G)^s$, $\mathcal{C}_H = (H,\, \mathsf{var}(H),\, d \cdot 2,\, e_H,\, K,\, O,\, \delta_H)^s$. Unit, Decide, BackTrue, and BackFalse: Apart from the additional elements V, d, e and the component state s, the rules are defined as in the former calculus [34]. The arguments given in the proof there apply here as well, and after applying rules Unit, Decide, BackTrue, or BackFalse, Inv. (1) - (5) hold. Notice that in the proof of Inv. (4), it suffices to replace "DSOP" by "d-DNNF", since the relevant property here is determinism. Since $e' = 0$, Inv. (6) and (7) do not apply. An open state is mapped to an open state, hence Inv. (8) holds.

CompTrue and CompFalse: Invariants (1) and (2) hold, since I remains unaffected. Since $\mathcal{C}'$ is closed, Inv. (3) and (4) are met. The proof that Inv. (5) holds is carried

out similarly to the proof of Proposition 1 in our previous work [34] for rules EndTrue and EndFalse, respectively. Since $e' = 0$ and $I' = I$, Inv. (6) - (8) hold.

Decompose: The parent component $\mathcal{C}$ remains unaltered except for $e' = 2$ and for its state, which becomes f. Both its subcomponents $\mathcal{C}_G$ and $\mathcal{C}_H$ are open, and we have $J_G = J_H = \varepsilon$ and $e_G = e_H = 0$. Therefore, Inv. (1) - (5) hold. Invariant (6) is satisfied by the definition of rule Decompose. Since $\mathcal{C}'$ is decomposed and $\mathcal{C}_G$ and $\mathcal{C}_H$ are open by definition, Inv. (7) and (8) hold as well.

ComposeBack: It suffices to show that the validity of the invariants for $\mathcal{C}'$ is preserved, since $\mathcal{C}_G$ and $\mathcal{C}_H$ do not occur in the target state. The most recent decision literal is flipped, similar to rule BackTrue. The same argument to the one given there applies, and Inv. (1) and (2) are satisfied. We need to show that $F \wedge \neg(M \vee (I \wedge N \wedge O)) \wedge \mathsf{decs}_{\leqslant n}(PK\ell) \models (PK\ell)_{\leqslant n}$ holds for all n. The decision levels of the literals in PK do not change, except for the one of ℓ, which is decremented from $e+1$ to e. The literal ℓ also stops from being a decision literal. Since $\delta(PK\ell) = e$, we can assume $n \leq e$. Furthermore, $F \wedge \neg(M \vee (I \wedge N \wedge O)) \wedge \mathsf{decs}_{\leqslant n}(PK\ell)) \equiv (\neg I \wedge (F \wedge \neg M \wedge \mathsf{decs}_{\leqslant n}(I))) \vee (F \wedge \neg M \wedge \neg(N \wedge O) \wedge \mathsf{decs}_{\leqslant n}(I))$, since ℓ is not a decision literal in $PK\ell$ and $I_{\leqslant e} = PK$ and thus $I_{\leqslant n} = (PK)_{\leqslant n}$ by definition. By applying the induction hypothesis, we get $\neg I \wedge F \wedge \neg M \wedge \mathsf{decs}_{\leqslant n}(PK\ell) \models (PK)_{\leqslant n}$, and hence $F \wedge \neg(M \vee (I \wedge N \wedge O)) \wedge \mathsf{decs}_{\leqslant n}(PK\ell) \models (PK)_{\leqslant n}$. We still need to show that $F \wedge \neg(M \vee (I \wedge N \wedge O)) \wedge \mathsf{decs}_{\leqslant e}(PK\ell) \models \ell$, as $\delta(\ell) = e$ in $PK\ell$ after applying ComposeBack and thus ℓ disappears from the proof obligation for $n < e$. Notice that $F \wedge \neg D \models I$ using again the induction hypothesis for $n = e + 1$. This gives us $F \wedge \neg\mathsf{decs}_{\leqslant e}(PK) \wedge \neg\ell \models I$ and thus $F \wedge \neg\mathsf{decs}_{\leqslant e}(PK) \wedge \neg I \models \ell$ by conditional contraposition, and Inv. (3) holds.

For proving that Inv. (4) holds, we consider two cases: (A) $I \wedge N \wedge O \neq \bot$, i.e., there exists an extension of I which satisfies F, and (B) $I \wedge N \wedge O = \bot$, i.e., all extensions of I falsify F. For both cases, we know that $I \vee O(I)$ is a d-DNNF.

(A) We need to show that $M \vee (I \wedge N \wedge O) \vee O(PK\ell)$ is a d-DNNF. Due to $\delta(I) = e + 1$, we have $O(I) = I \vee R_{\leqslant e+1}(I) = I \vee R_{\leqslant e}(I) \vee R_{=e+1}(I)$. The pending search space of $PK\ell$ is given by $O(PK\ell) = PK\ell \vee R_{\leqslant e}(PK\ell)$. But $PK = I_{\leqslant e}$ and $PK\ell = I_{\leqslant e}\ell = R_{=e+1}(I)$, since $\neg\ell \in \mathsf{decs}(I)$ and $\delta(\neg\ell) = e + 1$. Furthermore, $R_{\leqslant e}(PK\ell) = R_{\leqslant e}(PK)$, since $\ell \notin \mathsf{decs}(PK\ell)$ and $\delta(\ell) = e$, hence $R_{\leqslant e}(PK\ell) = R_{\leqslant e}(I)$. We have $O(PK\ell) = R_{=e+1}(I) \vee R_{\leqslant e}(I)$, hence $O(PK\ell) \vee I = O(I)$ and $(M \vee I) \vee O(PK\ell) = M \vee O(I)$, which is a DSOP and hence a d-DNNF. Now I, N, and O are defined over pairwise disjoint sets of variables by construction, i.e., $I \wedge N \wedge O$ is decomposable, and $M \vee (I \wedge N \wedge O) \vee O(PK\ell)$ is a d-DNNF.

(B) We need to show that $M \vee O(PK\ell)$ is a d-DNNF. As just shown, $O(PK\ell) \vee I = O(I)$. Now $M \vee O(PK\ell) = M \vee R_{\leqslant e+1}(I)$. Recalling that $R_{\leqslant e+1}(I)$ is equal to $O(I)$ without I and $M \vee O(I)$ is a d-DNNF by the premise, $M \vee O(PK\ell)$ is a d-DNNF as well. Therefore, Inv. (4) holds.

For the proof of the validity of Inv. (5), given $M \vee F \wedge O(I) \equiv F$, the same two cases are relevant: (A) $I \wedge N \wedge O \neq \bot$ and (B) $I \wedge N \wedge O = \bot$.

(A) We have to show that $M \vee (I \wedge N \wedge O) \vee (F \wedge O(PK\ell)) \equiv F$. From $O(PK\ell) \vee I = O(I)$ we get $M \vee (F \wedge O(I)) = M \vee (F \wedge (O(PK\ell)) \vee I) =$

$M \vee (F \wedge O(PK\ell)) \vee (F \wedge I) \equiv F$. But $F \wedge I \equiv I \wedge N \wedge O$. Therefore $M \vee (F \wedge O(I)) \equiv M \vee (F \wedge O(PK\ell)) \vee (I \wedge N \wedge O) = M \vee (I \wedge N \wedge O) \vee (F \wedge O(PK\ell)) \equiv F$.

(B) We must show that $M \vee (F \wedge O(PK\ell)) \equiv F$. Similarly to (A) we have $M \vee (F \wedge O(I)) \equiv M \vee (F \wedge O(PK\ell)) \vee (F \wedge I) \equiv M \vee (F \wedge O(PK\ell)) \equiv F$, due to $F \wedge I \equiv F$. Therefore, Inv. (5) holds after applying rule ComposeBack. We have $e' = 0$, and $\mathcal{C}'$ is open, hence Inv. (6) - (8) trivially hold.

ComposeEnd: It suffices to show that after applying rule ComposeBack the invariants are met by $\mathcal{C}'$, since its subcomponent states $\mathcal{C}_G$ and $\mathcal{C}_H$ do not occur in the target state anymore. Due to $I' = I$ and $\mathsf{decs}(I) = \emptyset$ and since $\mathcal{C}'$ is closed, Inv. (1) - (4) trivially hold.

For proving that invariant (5) holds after applying rule ComposeEnd, i.e., that $M \vee (I \wedge N \wedge O) \vee (F \wedge O(I)) \equiv F$, the same two cases need to be distinguished: (A) $I \wedge N \wedge O \neq \bot$ and (B) $I \wedge N \wedge O = \bot$.

(A) From $\mathsf{decs}(I) = \emptyset$, we get $O(I) = I$ and $F \wedge O(I) = F \wedge I$. Recalling that $F \wedge I \equiv I \wedge N \wedge O$, we obtain $M \vee (I \wedge N \wedge O) \vee (F \wedge O(I)) \equiv M \vee (F \wedge O(I)) \equiv F$ by the premise.

(B) We have $M \vee (I \wedge N \wedge O) \vee (F \wedge O(I)) = M \vee (F \wedge O(I)) \equiv F$ by the premise, and Inv. (5) holds after executing rule ComposeEnd. Invariants (6) - (8) trivially hold, due to $e' = 0$ and $I' = I$ and hence $\mathsf{decs}(I') = \emptyset$.

Corollary 1 (Soundness of ACD Run). ACD *starting with an initial global state is sound.*

Proof. The initial state is sound by Lemma 1, and all rule applications lead to a sound state according to Theorem 1.

Lemma 2 (Correctness of Closed Component State). *For any closed component* $(F, V, d, 0, I, M, \delta)^c$ *it holds that* $M \equiv F$.

Proof. Follows from Theorem 1, proof of Inv. (5) for rules CompTrue, CompFalse, and ComposeEnd, which are the only rules closing a component.

Theorem 2 (Correctness of Final Global State). *In the final global state* $\mathcal{S}_n = \{(F, V, d, 0, I, M, \delta)^c\}$ *of* ACD*,* $M \equiv F$ *holds.*

Proof. Correctness of the closed root component follows from Lemma 2. We need to show that the final global state contains exactly the closed root component. The initial global state consists of the open root component. Additional components are created exclusively by rule Decompose, and a parent component state can only be closed by rule ComposeEnd, which also removes its subcomponents from the global state. Hence the root component can only be closed if it has no subcomponents. But since the initial global state contains exclusively the root component, the final global state contains only the closed root component.

Theorem 3 (Progress). ACD *always makes progress.*

Unit:
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 2, 2, \ldots, 2], o)\}$
$\succ_{\text{ACD}}$
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 0, 2, \ldots, 2], o)\}$

Decide:
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 2, 2, \ldots, 2], o)\}$
$\succ_{\text{ACD}}$
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 1, 2, \ldots, 2], o)\}$

BackTrue:
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 1, l_{k+2}, \ldots, l_{|V|}], o)\}$
$\succ_{\text{ACD}}$
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 0, l'_{k+2}, \ldots, l'_{|V|}], o)\}$

BackFalse:
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 1, l_{k+2}, \ldots, l_{|V|}], o)\}$
$\succ_{\text{ACD}}$
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 0, l'_{k+2}, \ldots, l'_{|V|}], o)\}$

CompTrue:
$\mathcal{S}' \uplus \{(d, t, o)\} \succ_{\text{ACD}} \mathcal{S}' \uplus \{(d, t, c)\}$

CompFalse:
$\mathcal{S}' \uplus \{(d, t, o)\} \succ_{\text{ACD}} \mathcal{S}' \uplus \{(d, t, c)\}$

Decompose:
$\mathcal{S}' \uplus \{(d, t, o)\} \succ_{\text{ACD}} \mathcal{S}' \uplus \{(d, t, f), (d \cdot 1, [2, \ldots, 2], o), (d \cdot 2, [2, \ldots, 2], o)\}$

ComposeBack:
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 1, l_{k+2}, \ldots, l_{|V|}], f), (d \cdot 1, t_1, c), (d \cdot 2, t_2, c)\}$
$\succ_{\text{ACD}}$
$\mathcal{S}' \uplus \{(d, [l_1, \ldots, l_k, 0, l'_{k+2}, \ldots, l'_{|V|}, o])\}$

ComposeEnd:
$\mathcal{S}' \uplus \{(d, t, f), (d \cdot 1, t_1, c), (d \cdot 2, t_2, c)\} \succ_{\text{ACD}} \mathcal{S}' \uplus \{(d, t, c)\}$

Fig. 4. Rule applications lead to smaller global states.

Proof. The proof is conducted by induction over the rules. We show that as long as the root component is not closed, a rule is applicable. For the case $\mathcal{S} \uplus \{\mathcal{C}\}$, where $\mathcal{C} = (F, V, d, 0, I, M, \delta)^o$ has no subcomponents, the proof is identical to the one showing progress in our previous work [34] replacing EndTrue with CompTrue and EndFalse with CompFalse, and by checking whether the preconditions for rule Decompose are met if rule Unit is not applicable and before taking a decision. Now let the global state be given by $\mathcal{S} \uplus \{\mathcal{C}\}$ where $\mathcal{C} = (F, V, d, 2, I, M, \delta)^f$ is decomposed. Due to Inv. (6), $\mathcal{S}$ contains $\mathcal{C}_G = (G, \mathsf{var}(G), d \cdot 1, e_G, J_G, N, \delta_G)^s$ and $\mathcal{C}_H = (H, \mathsf{var}(H), d \cdot 2, e_H, J_H, O, \delta_H)^s$ such that $F|_I = G \wedge H$ and $\mathsf{var}(G) \cap \mathsf{var}(H) = \emptyset$. Assume $s = c$ for both $\mathcal{C}_G$ and $\mathcal{C}_H$. If $\mathsf{decs}(I) = \emptyset$, rule ComposeEnd is applicable. Otherwise, similarly to rule BackTrue, we can show that all preconditions of rule ComposeBack are met. If instead $s \in \{f, o\}$ for at least one of $\mathcal{C}_G$ and $\mathcal{C}_H$, the non-closed component(s) are processed further, and as soon as both $\mathcal{C}_G$ and $\mathcal{C}_H$ are closed, rule ComposeEnd or ComposeBack can be applied. This proves that ACD always makes progress.

Theorem 4 (Termination). ACD *always terminates.*

Proof. We need to show that no infinite sequence of rule applications can happen. To this end, we define a strict, well-founded ordering $\succ_{\text{ACD}}$ on the global states and show that $\mathcal{S} \leadsto_{\mathsf{R}} \mathcal{T}$ implies $\mathcal{S} \succ_{\text{ACD}} \mathcal{T}$ for all $\mathcal{S}, \mathcal{T} \in S$ and rules R in ACD. Global states are sets of components, and $\succ_{\text{ACD}}$ is the multiset extension of a component ordering $\succ_{\mathsf{c}} = (\succ_{\mathsf{cl}}, \succ_{\mathsf{tr}}, \succ_{\mathsf{cs}})$, where $\succ_{\mathsf{cl}}$, $\succ_{\mathsf{tr}}$, and $\succ_{\mathsf{cs}}$ are orderings on component levels, trails, and component states, respectively. We want to compare trails defined over the same set of variables V, and to this end we represent them

DecomposeG:	$\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M,\, \delta)^o\} \leadsto_{\mathsf{DecomposeG}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, n,\, I,\, M,\, \delta)^f, ((G_i,\, U_i,\, d \cdot i,\, 0,\, \varepsilon,\, \bot,\, \infty)^o)_{i=1}^n,\}$ if $F\|_I \neq \top$ and $\bot \notin F\|_I$ and $\mathsf{units}(F\|_I) = \emptyset$ and $\bigwedge_{i=1}^n G_i \stackrel{\text{def}}{=} F\|_I$ and $n \geq 2$ and $U_i \stackrel{\text{def}}{=} \mathsf{var}(G_i)$ and $U_i \cap U_j = \emptyset$ for $1 \leq i,j \leq n$ and $i \neq j$
ComposeBackG:	$\mathcal{S} \uplus \{(F,\, V,\, d,\, n,\, I,\, M,\, \delta)^f, ((G_i,\, U_i,\, d \cdot i,\, 0,\, J_i,\, N_i,\, \delta_i)^c)_{i=1}^n\} \leadsto_{\mathsf{ComposeBackG}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, P K \ell,\, M \vee I \wedge N,\, \delta[L \mapsto \infty][\ell \mapsto e])^o\}$ if $PQ \stackrel{\text{def}}{=} I$ and $D \stackrel{\text{def}}{=} \neg\mathsf{decs}(I)$ and $e+1 \stackrel{\text{def}}{=} \delta(D \setminus \{\ell\}) = \delta(I)$ and $\ell \in D$ and $e = \delta(D \setminus \{\ell\}) = \delta(P)$ and $K \stackrel{\text{def}}{=} Q_{\leqslant e}$ and $L \stackrel{\text{def}}{=} Q_{>e}$ and $N \stackrel{\text{def}}{=} \bigwedge_{i=1}^n N_i$
ComposeEndG:	$\mathcal{S} \uplus \{(F,\, V,\, d,\, n,\, I,\, M,\, \delta)^f, ((\top,\, U_i,\, d \cdot i,\, 0,\, J_i,\, N_i,\, \delta_i)^c)_{i=1}^n\} \leadsto_{\mathsf{ComposeEndG}}$ $\mathcal{S} \uplus \{(F,\, V,\, d,\, 0,\, I,\, M \vee I \wedge N,\, \delta[I \mapsto \delta])^c\}$ if $\mathsf{decs}(I) = \emptyset$ and $N \stackrel{\text{def}}{=} \bigwedge_{i=1}^n N_i$

Fig. 5. Generalized transition rules.

as lists over $\{0, 1, 2\}$. A trail $I = \ell_1 \dots \ell_k$ defined over V, where $k \leq |V|$, is represented as $[l_1, \dots, l_k, 2, \dots, 2]$, where $l_i = 0$ if ℓ_i is a propagation literal and $l_i = 1$ if ℓ_i is a decision literal. The last $|V| - m$ positions with value 2 represent the unassigned variables. Trails defined over the same variable set are encoded into lists of the same length. This representation induces a lexicographic order $>_{\mathsf{lex}}$ on trails, and we define $\succ_{\mathsf{tr}}$ as the restriction of $>_{\mathsf{lex}}$ to $\{[l_1, \dots, l_{|V|}] \mid l_i \in \{0, 1, 2\}$ for $1 \leq i \leq |V|\}$, i.e., we have $t_1 \succ_{\mathsf{tr}} t_2$ if $t_1 >_{\mathsf{lex}} t_2$. The ordering $\succ_{\mathsf{tr}}$ is well-founded, its minimal element is $[0, \dots, 0]$. The component state takes values in $\{o, f, c\}$, and we define $\succ_{\mathsf{cs}}$ as $>_{\mathsf{lex}}$, i.e., $s_1 \succ_{\mathsf{cs}} s_2$ if $s_1 >_{\mathsf{lex}} s_2$. The minimal element of $\succ_{\mathsf{cs}}$ is c, hence $\succ_{\mathsf{cs}}$ is well-founded. Given two component levels d_1 and d_2, we define $d_1 \succ_{\mathsf{cl}} d_2$ if $\mathsf{length}(d_1) < \mathsf{length}(d_2)$. This may seem counterintuitive but is needed to ensure that the execution of rule Decompose results in a smaller state, since both the component state and the trail of the new subcomponents are of higher order than those of their parent. To see that $\succ_{\mathsf{cs}}$ is well-founded, recall that we consider finite variable sets. Their size provides an upper limit on the length of the component level representation and a minimal element of $\succ_{\mathsf{cs}}$.

Now we define the component ordering $\succ_{\mathsf{c}} = (\succ_{\mathsf{cl}}, \succ_{\mathsf{tr}}, \succ_{\mathsf{cs}})$. Let two components be $\mathcal{C}_1 = (d_1, t_1, s_1)$ and $\mathcal{C}_2 = (d_2, t_2, s_2)$. We have $\mathcal{C}_1 \succ_{\mathsf{c}} \mathcal{C}_2$ if $\mathcal{C}_1 \neq \mathcal{C}_2$ and $d_1 \succ_{\mathsf{cl}} d_2$ or $d_1 = d_2$ and either $t_1 \succ_{\mathsf{tr}} t_2$ or $t_1 = t_2$ and $s_1 \succ_{\mathsf{cs}} s_2$. Clearly $\succ_{\mathsf{c}}$ is well-founded, since $\succ_{\mathsf{tr}}$, $\succ_{\mathsf{cs}}$, and $\succ_{\mathsf{cl}}$ are well-founded. For two global states $\mathcal{S}$ and $\mathcal{T}$, we have $\mathcal{S} \succ_{\mathrm{ACD}} \mathcal{T}$ if $\mathcal{S} \neq \mathcal{T}$ and for each component $\mathcal{C}$ such that $\mathcal{C}$ is larger in $\mathcal{T}$ than in $\mathcal{S}$ with respect to $\succ_{\mathsf{c}}$, $\mathcal{S}$ contains a component $\mathcal{C}'$ that is larger in $\mathcal{S}$ than in $\mathcal{T}$. Since $\succ_{\mathsf{c}}$ is well-founded, also $\succ_{\mathrm{ACD}}$ is well-founded. Figure 4 shows that each rule application leads to a smaller global state, concluding our proof.

6 Generalization

The generalized rules are listed in Fig. 5. In our generalized framework, we have $F|_I = \bigwedge_{i=1}^n G_i$, and $\mathsf{var}(G_i) \cap \mathsf{var}(G_j) = \emptyset$ for $i, j \in \{1, \dots, n\}$ and $i \neq j$ (rule DecomposeG). Similarly to their equivalents in ACD, rules ComposeBackG and ComposeEndG are applicable if all subcomponents are closed.

7 Discussion

We have presented Abstract CNF2dDNNF, or ACD for short, a formal framework for compiling a formula in CNF into d-DNNF combining CDCL-based model enumeration with chronological backtracking [34] and dynamic component analysis [4]. Conflict-driven clause learning enables our framework to escape regions without solution early, and chronological backtracking prevents multiple model enumeration without the need for remembering already found models using blocking clauses, which slow down unit propagation. However, the absence of blocking clauses also prevents the use of restarts. If exclusively the rules Unit, Decide, BackTrue, BackFalse, CompTrue, and CompFalse are used, a DSOP representation of F is computed. Unit propagation is prioritized due to its potential to reduce the number of decisions and thus of right branches to be explored. Favoring decompositions over decisions may also shrink a larger part of the search space. Our framework lays the theoretical foundation for practical All-SAT and #SAT solving based on chronological CDCL. Any implementation which can be modeled by ACD exhibits its properties, in particular its correctness, which has been established in a formal proof.

Comparison with Available Tools. There exist other knowledge compilers addressing d-DNNFs. We want to mention c2d [20], Dsharp [37], and D4 [30], which also execute an exhaustive search and conflict analysis. However, our approach differs conceptually from these tools in several ways. The most prominent ones are the use of CDCL with chronological backtracking [33,38] instead of CDCL with non-chronological backtracking and the way the d-DNNF is created. Our method generates DSOP representations of formulae which can not be decomposed further by an exhaustive (partial) model enumeration and then combines the result, while the tools mentioned above generate the d-DNNF by recording the execution trace as a graph [26,27]. As ACD, both D4 and Dsharp adopt a dynamic decomposition strategy, while c2d constructs a decomposition tree which it then uses for for component analysis.

Future Research Directions. We plan to implement a proof of concept of our calculus in order to compare the size of the returned d-DNNF with the ones obtained by c2d, D4, and Dsharp. For dynamic component analysis, one could follow the algorithm implemented in COMPSAT [6], while dual reasoning [32] and logical entailment [35] enable the detection of short partial models. This is particularly interesting in tasks where the length of the d-DNNF is crucial. Dual reasoning has shown to be almost competitive on CNFs if the search space is small, we therefore expect that component analysis boosts its performance. The major challenge posed by the second approach lies in an efficient implementation of the oracle calls required by the entailment checks. It would be interesting to investigate the impact of dynamic component analysis on a recent implementation [46] of model enumeration by chronological CDCL [34]. Cache structures, being an inherent part of modern knowledge compilers and #SAT solvers [11,16,19,20,30,31,37,41,42,47,49] due to their positive impact on solver efficiency [1], should be added to any implementation of our framework. Finally,

an important research topic is that of optimizing the encoding of a formula making best use of component analysis [14]. Related to this question is whether formulae stemming from practical applications are decomposable in general.

Acknowledgements. My thanks go to Armin Biere for a fruitful discussion when I got stuck in a first, very raw version of the proof, and to Martin Bromberger for his input enhancing it.

References

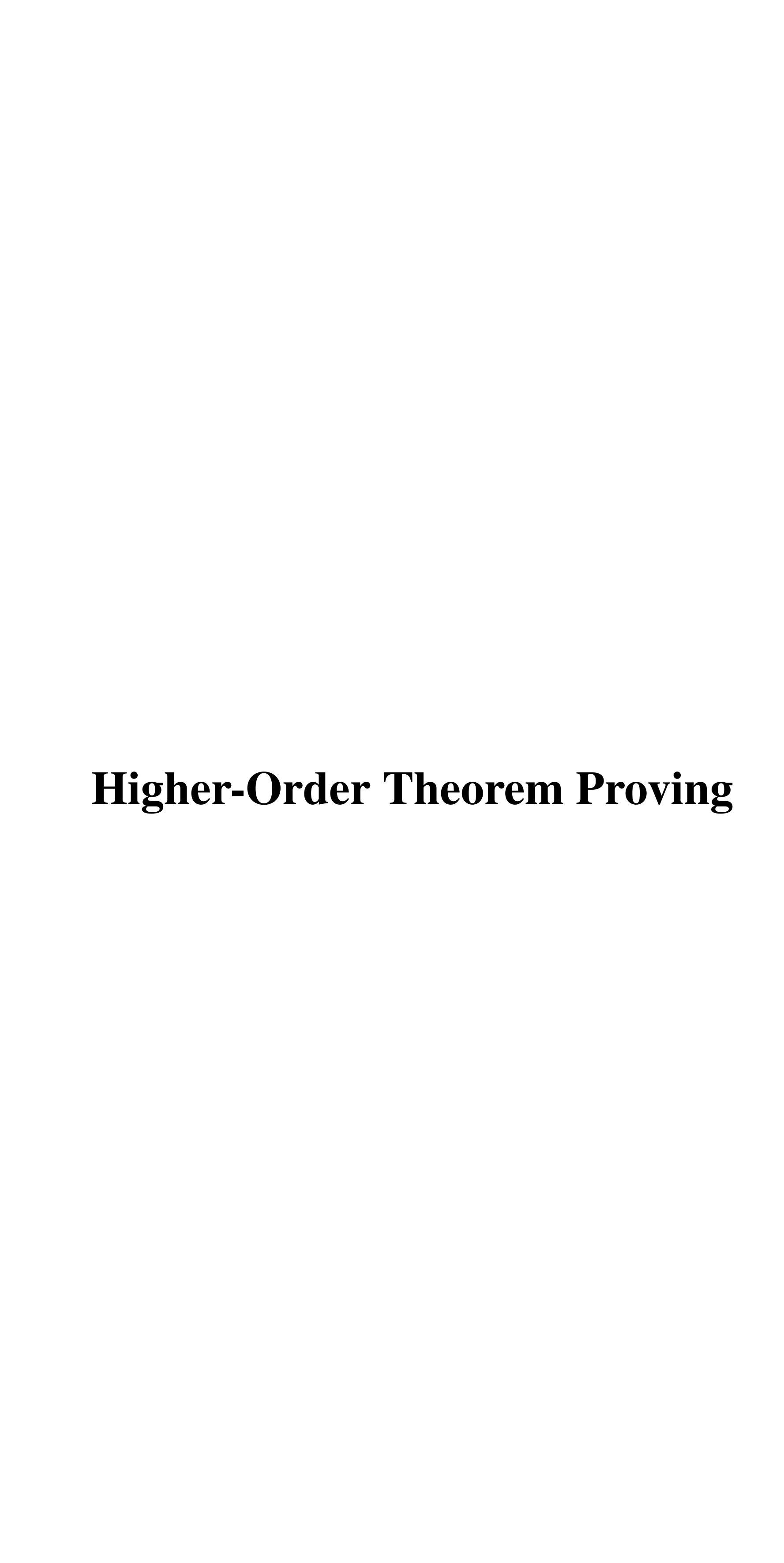
1. Bacchus, F., Dalmao, S., Pitassi, T.: DPLL with caching: a new algorithm for #SAT and Bayesian inference. Electron. Colloquium Comput. Complex. **TR03-003** (2003)
2. Barrett, A.: From hybrid systems to universal plans via domain compilation. In: ICAPS, pp. 44–51. AAAI (2004)
3. Barrett, A.: Model compilation for real-time planning and diagnosis with feedback. In: IJCAI, pp. 1195–1200. Professional Book Center (2005)
4. Bayardo Jr., R., Pehoushek, J.: Counting models using connected components. In: AAAI/IAAI, pp. 157–162. AAAI Press/The MIT Press (2000)
5. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L.: Compact DSOP and partial DSOP forms. Theory Comput. Syst. **53**(4), 583–608 (2013)
6. Biere, A., Sinz, C.: Decomposing SAT problems into connected components. J. Satisf. Boolean Model. Comput. **2**(1–4), 201–208 (2006)
7. Bollig, B., Buttkus, M.: On limitations of structured (deterministic) DNNFs. Theory Comput. Syst. **64**(5), 799–825 (2020)
8. Bollig, B., Farenholtz, M.: On the relation between structured d-DNNFs and SDDs. Theory Comput. Syst. **65**(2), 274–295 (2021)
9. Bova, S., Capelli, F., Mengel, S., Slivovsky, F.: On compiling CNFs into structured deterministic DNNFs. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 199–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_15
10. Bova, S., Capelli, F., Mengel, S., Slivovsky, F.: Knowledge compilation meets communication complexity. In: IJCAI, pp. 1008–1014. IJCAI/AAAI Press (2016)
11. Burchard, J., Schubert, T., Becker, B.: Laissez-faire caching for parallel #SAT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 46–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_5
12. Burchard, J., Schubert, T., Becker, B.: Distributed parallel #SAT solving. In: CLUSTER, pp. 326–335. IEEE Computer Society (2016)
13. Cadoli, M., Donini, F.M.: A survey on knowledge compilation. AI Commun. **10**(3–4), 137–150 (1997)
14. Chavira, M., Darwiche, A.: Encoding CNFs to empower component analysis. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 61–74. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_9
15. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational Bayesian networks for exact inference. Int. J. Approx. Reason. **42**(1–2), 4–20 (2006)
16. Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for Boolean satisfiability solvers. J. Satisf. Boolean Model. Comput. **6**(1–3), 99–120 (2009)
17. Darwiche, A.: Compiling knowledge into decomposable negation normal form. In: IJCAI, pp. 284–289. Morgan Kaufmann (1999)

18. Darwiche, A.: Decomposable negation normal norm. J. ACM **48**(4), 608–647 (2001)
19. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. Appl. Non Class. Logics **11**(1–2), 11–34 (2001)
20. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: ECAI, pp. 328–332. IOS Press (2004)
21. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: IJCAI, pp. 819–826. IJCAI/AAAI (2011)
22. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002)
23. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
24. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
25. Fargier, H., Mengin, J.: A knowledge compilation map for conditional preference statements-based languages. In: AAMAS, pp. 492–500. ACM (2021)
26. Huang, J., Darwiche, A.: DPLL with a trace: From SAT to knowledge compilation. In: IJCAI, pp. 156–162. Professional Book Center (2005)
27. Huang, J., Darwiche, A.: The language of search. J. Artif. Intell. Res. **29**, 191–219 (2007)
28. Huang, X., Izza, Y., Ignatiev, A., Cooper, M.C., Asher, N., Marques-Silva, J.: Tractable explanations for d-DNNF classifiers. In: AAAI, pp. 5719–5728. AAAI Press (2022)
29. Koriche, F., Lagniez, J., Marquis, P., Thomas, S.: Knowledge compilation for model counting: affine decision trees. In: IJCAI, pp. 947–953. IJCAI/AAAI (2013)
30. Lagniez, J., Marquis, P.: An improved Decision-DNNF compiler. In: IJCAI, pp. 667–673. ijcai.org (2017)
31. Lagniez, J., Marquis, P., Szczepanski, N.: DMC: a distributed model counter. In: IJCAI, pp. 1331–1338. ijcai.org (2018)
32. Möhle, S., Biere, A.: Dualizing projected model counting. In: ICTAI, pp. 702–709. IEEE (2018)
33. Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 250–266. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_18
34. Möhle, S., Biere, A.: Combining conflict-driven clause learning and chronological backtracking for propositional model counting. In: GCAI. EPiC Series in Computing, vol. 65, pp. 113–126. EasyChair (2019)
35. Möhle, S., Sebastiani, R., Biere, A.: Four flavors of entailment. In: Pulina, L., Seidl, M. (eds.) SAT 2020. LNCS, vol. 12178, pp. 62–71. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_5
36. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM (2001)
37. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) AI 2012. LNCS (LNAI), vol. 7310, pp. 356–361. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30353-1_36
38. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 111–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_7

39. Palacios, H., Bonet, B., Darwiche, A., Geffner, H.: Pruning conformant plans by counting models on compiled d-DNNF representations. In: ICAPS, pp. 141–150. AAAI (2005)
40. Pipatsrisawat, K., Darwiche, A.: A new d-DNNF-based bound computation algorithm for functional E-MAJSAT. In: IJCAI, pp. 590–595 (2009)
41. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT (2004)
42. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: a scalable probabilistic exact model counter. In: IJCAI, pp. 1169–1176. ijcai.org (2019)
43. Siddiqi, S.A., Huang, J.: Probabilistic sequential diagnosis by compilation. In: ISAIM (2008)
44. Marques-Silva, J.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227. IEEE Computer Society/ACM (1996)
45. Marques-Silva, J.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
46. Spallitta, G., Sebastiani, R., Biere, A.: Enumerating disjoint partial models without blocking clauses. CoRR abs/2306.00461 (2023)
47. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_38
48. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Compiling CP subproblems to MDDs and d-DNNFs. Constraints An Int. J. **24**(1), 56–93 (2019)
49. Zhang, L., Malik, S.: Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 287–298. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_22

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.



Higher-Order Theorem Proving

Hammering Floating-Point Arithmetic

Olle Torstensson[1] and Tjark Weber[2(✉)]

[1] Linköping University, Linköping, Sweden
olle.torstensson@liu.se
[2] Uppsala University, Uppsala, Sweden
tjark.weber@it.uu.se

Abstract. Sledgehammer, a component of the interactive proof assistant Isabelle/HOL, aims to increase proof automation by automatically discharging proof goals with the help of external provers. Among these provers are a group of satisfiability modulo theories (SMT) solvers with support for the SMT-LIB input language. Despite existing formalizations of IEEE floating-point arithmetic in both Isabelle/HOL and SMT-LIB, Sledgehammer employs an abstract translation of floating-point types and constants, depriving the SMT solvers of the opportunity to make use of their dedicated decision procedures for floating-point arithmetic.

We show that, by extending Sledgehammer's translation from the language of Isabelle/HOL into SMT-LIB with an interpretation of floating-point types and constants, floating-point reasoning in SMT solvers can be made available to Isabelle/HOL. Our main contribution is a description and implementation of such an extension. An evaluation of the extended translation shows a significant increase of Sledgehammer's success rate on proof goals involving floating-point arithmetic.

1 Introduction

Interactive theorem proving is one of the more flexible and powerful formal verification techniques available. However, finding a proof outline with intermediate proof steps just simple enough for a proof assistant to be able to discharge automatically may require a considerable amount of time and effort, even from a seasoned user. As an example, the seL4 micro-kernel, the product of about two person-years and 9000 lines of code, took a total of about 20 person-years and 200,000 lines of proof development to formally verify [29]. For this reason, increasing proof automation in interactive proof assistants is crucial to further broaden their applicability.

As a way of tackling this issue, many interactive proof assistants have the ability to transfer the proof burden of some of the intermediate steps onto *automated* reasoning systems with automatic proof methods better suited for the task. This approach has proven to be quite successful in bringing the number of required user interactions down for many types of problems, thus increasing productivity.

Among these proof assistants, we find Isabelle/HOL [34] and its powerful proof-delegation tool Sledgehammer [36], which acts as an interface between

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 217–235, 2023.
https://doi.org/10.1007/978-3-031-43369-6_12

Isabelle/HOL and a number of external provers. In addition to traditional (resolution-based) first-order automated theorem provers (ATPs) such as E [40], SPASS [45] and Vampire [38] and the higher-order ATP Zipperposition [9], these external provers include satisfiability modulo theories (SMT) solvers such as CVC4 [7], veriT [15] and Z3 [31]. SMT solvers are highly specialized for reasoning within certain logical theories (e.g., integers, real numbers, and bit vectors), and often implement decision procedures more efficient than those found in the automatic proof methods of Isabelle/HOL.

Whether an external prover succeeds in solving a delegated proof obligation depends, among other factors, on how the proof obligation is encoded in the language of the prover. SMT solvers support the SMT-LIB input language [6], which offers both uninterpreted (free) type and function symbols that are declared by the user, as well as theory-specific *interpreted* types and operations that have a fixed semantics. Dedicated inference rules and decision procedures for specific theories that are available in SMT solvers are typically employed only when the types and operations that appear in the delegated proof obligation are interpreted. An abstract translation that leaves types and operations uninterpreted will deprive external solvers of the opportunity to make use of their dedicated decision procedures for specific background theories, and will instead have to rely on a sufficient set of facts being passed to the solver along with the proof obligation.

One of the more recent additions to the growing set of theories supported by major SMT solvers is that of floating-point arithmetic [16]. A formalization of IEEE floating-point arithmetic in Isabelle/HOL has been available in the Archive of Formal Proofs for nearly a decade [46]. However, Sledgehammer has not yet caught up to this development; its SMT component does not implement an interpretation of floating-point types and operations. Our aim is to provide such an interpretation, with the purpose of increasing the success rate for floating-point proof obligations delegated to SMT solvers, and thereby to increase the degree of automation in the interactive proof process.

As an example, let us consider the commutativity of floating-point addition. SMT solvers that support floating-point arithmetic typically have no trouble proving that `x + y = y + x` when they can assume that `x` and `y` denote floating-point numbers, and that `+` denotes IEEE floating-point addition (i.e., when `+` is translated as `fp.add`). However, if this formula is translated in an uninterpreted fashion, the problem becomes much harder: it now requires to show commutativity of a user-declared function over a user-declared type. Whether the SMT solver will succeed in this case depends on many factors, including which additional facts (definitions and lemmas) are passed along from the interactive proof assistant together with the proof obligation itself.

Contributions. We define a formal model of floating-point arithmetic in Isabelle/HOL that implements the SMT-LIB floating-point theory (Sect. 3).

We then extend the SMT solver integration in Isabelle/HOL by adding support for floating-point arithmetic, i.e., by treating floating-point types and operations as interpreted in the translation from the language of Isabelle/HOL to the SMT-LIB input format. In addition to describing this extension in

detail (Sect. 4), we provide an implementation (in the Archive of Formal Proofs [46]) that supports Sledgehammer. To the best of our knowledge, this makes Isabelle/HOL the first interactive proof assistant to employ an interpreted translation for floating-point arithmetic in its integration of automated theorem provers.

An evaluation (Sect. 5), performed on a representative set of floating-point proof obligations from interactive proof, confirms the expectation that our translation extension significantly increases Sledgehammer's success rate on proof goals involving floating-point arithmetic, albeit at the cost of lower success rates for proof reconstruction—at this stage, our integration typically requires the external SMT solvers to be trusted as *oracles*.

2 Background

In this section, we cover additional background information regarding Sledgehammer and floating-point arithmetic.

2.1 The Sledgehammer Proof Process

When trying to prove a conjecture in Isabelle, a user may, via a simple call to Sledgehammer, pass along the proof obligation to several external provers, which will then work on the problem in parallel. The statement to be proven is used by a relevance filter [30] to select additional facts (axioms and previously proven statements) that may help in finding a proof. All of these statements are then translated and compiled into a file in the input format of the external prover (in the case of SMT solvers, an SMT-LIB input file), as illustrated in Fig. 1.

After working on the problem, the external prover (if it does not time out) returns to Isabelle with its findings. At this point, if a prover reported the conjecture to be true, the user can either choose to view the prover as an *oracle* and accept the conjecture as a theorem (the dashed path in Fig. 1), or make Isabelle try to automatically reconstruct the proof internally, based on the additional facts sent with the conjecture and any proof details the prover may provide. Theorems that are only proved externally are marked with an *oracle* tag, meant to convey a certain amount of skepticism—reconstructed proofs are generally preferred, as they remove the consideration of possible bugs in the external prover, or in the translation between formats.

In Sledgehammer's translation module, types and constants are generally declared with a unique (freshly generated) identifier that has no inherent meaning to the external prover. A few Isabelle theories (e.g., those for integer arithmetic, real arithmetic, and bit vectors) define types and constants that are treated as interpreted by the translation into SMT-LIB [11], in which case they are mapped directly to their counterpart in the target logic—thereby allowing the SMT solvers to use their built-in decision procedures designed specifically to reason within the theories in question.

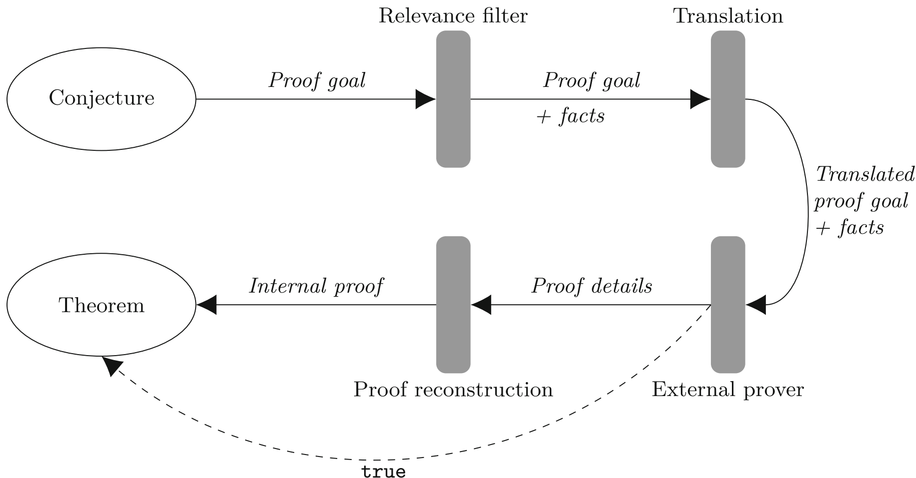


Fig. 1. A conjecture's journey to become a theorem via Sledgehammer

2.2 IEEE 754 Binary Floating-Point Arithmetic

The most common way to approximate the real numbers to a suitable finite set of numbers in modern hardware is via *floating-points*. Simulating real arithmetic using floating-points is not a straightforward task; the definitions of arithmetic operations are not always obvious, and should ideally not vary between implementations. To this end, the IEEE developed the technical standard IEEE 754 [26], aiming to provide clear specifications and recommendations on all aspects of floating-point arithmetic. To meet the needs of different applications, the standard specifies several floating-point *formats*, each defining a unique set of numbers.

A binary floating-point format is characterized by its exponent width $w \in \mathbb{N}$, and its precision $p \in \mathbb{N}$. A binary floating-point number, x, may then be represented in this format by a triple (s, e, f) of bit vectors of length 1, w, and $p-1$, respectively, such that (for finite x)

$$x = \begin{cases} (-1)^s \cdot 2^{1-\text{bias}(w)} \cdot (0 + \dfrac{f}{2^{p-1}}) & \text{if } \ \mathrm{e} = 0 \\ (-1)^s \cdot 2^{e-\text{bias}(w)} \cdot (1 + \dfrac{f}{2^{p-1}}) & \text{otherwise,} \end{cases} \tag{1}$$

where $\text{bias}(w) = 2^{w-1} - 1$. The standard also specifies two signed infinities, $+\infty$ and $-\infty$, denoting values that are too great in magnitude for the format. These are represented by the triples $(0, 1 \ldots 1, 0 \ldots 0)$ and $(1, 1 \ldots 1, 0 \ldots 0)$, respectively. Together, the sign s, the (biased) exponent e, and the fraction f constitute a unique representation of any finite or infinite floating-point number; in particular, the two numbers $+0$, represented by $(0, 0 \ldots 0, 0 \ldots 0)$, and -0, represented by $(1, 0 \ldots 0, 0 \ldots 0)$, are considered distinct. To represent the result

of invalid operations, such as 0/0, the standard defines a special *Not-a-Number* (NaN) value, represented via any triple $(s, 1 \dots 1, f)$ such that $f \neq 0 \dots 0$.[1]

Additionally, IEEE 754 specifies various arithmetic operations on floating-point numbers. Conceptually, floating-point arithmetic is carried out by converting floating-point numbers to more precise values, performing the corresponding arithmetic operation, and converting the result back to the original floating-point format, in an emulation of a rounded infinitely precise calculation. In an environment like Isabelle/HOL, where theories of real arithmetic are available, the task of carrying out calculations with infinite precision falls upon these, whereas the floating-point operations handle the rounding and special cases (e.g., an argument being NaN or infinite). IEEE 754 specifies precisely how this handling should be performed.

3 An Implementation of SMT-LIB Floating-Point Arithmetic in Isabelle/HOL

Formalizations of floating-point arithmetic are readily available for many proof assistants. For Isabelle/HOL, a formalization originally developed by Lei Yu is available from the Archive of Formal Proofs [46]. This defines a (polymorphic) type of floating-point numbers, whose instances correspond to IEEE floating-point formats with specific width and precision, and various arithmetic operations over this type.

However, although both are based on the IEEE standard, there are important semantic differences between this model and the SMT-LIB floating-point theory [16]. These differences would have rendered a direct interpretation of Lei Yu's model in the SMT-LIB floating-point theory unsound.

First, the SMT-LIB theory offers five rounding modes. The mode `roundNearestTiesToAway` (which is optional according to IEEE 754) was not available in the Isabelle/HOL model. Therefore, the enumerated type of rounding modes in Isabelle/HOL did not correspond to the `RoundingMode` sort in SMT-LIB. We have resolved this difference by adding support for `roundNearestTiesToAway` to Lei Yu's model. Although rounding is pervasive in IEEE—it is performed by most arithmetic operations—it is factored out into only two functions in the Isabelle/HOL model (`round` and `intround`), so that this was a relatively minor, local change.

Second, the formalization by Lei Yu emphasizes the bit representation of floating-point values (corresponding to specification level 4 in IEEE 754), while the SMT-LIB floating-point theory takes a more abstract view (corresponding to specification level 2 in IEEE 754). Specifically, in Lei Yu's formalization, each floating-point format contains multiple NaN values (with different bit representations), while the corresponding floating-point format in SMT-LIB only

[1] The IEEE 754 standard defines a *quiet* and a *signalling* NaN. This distinction is not present in the SMT-LIB floating-point theory, which is based on a higher level of abstraction.

contains a single (abstract) NaN value. To resolve this fundamental difference, we have constructed a new model of floating-point arithmetic in Isabelle/HOL. Our starting point is a quotient construction over the type `('e,'f) float` of floating-point numbers offered by Lei Yu's model. We first define an equivalence relation `is_nan_equivalent` on this type that relates all NaN values:

definition `is_nan_equivalent` :: `('e,'f) float` $\Rightarrow$ `('e,'f) float` $\Rightarrow$ `bool`
  where `is_nan_equivalent` $a\ b \ \equiv\ a = b \vee ($`is_nan` $a \wedge$ `is_nan` $b)$

We then define a new type `('e,'f) floatSingleNaN` that contains the equivalence classes of `('e,'f) float` with respect to the relation `is_nan_equivalent`:

quotient_type (overloaded) `('e,'f) floatSingleNaN` =
  `('e,'f) float` / `is_nan_equivalent`

The resulting type `('e,'f) floatSingleNaN` contains a single (abstract) NaN value. The (type) arguments `'e` and `'f` indicate the bit width of the exponent and fraction, respectively. A similar construction, but limited to the double-precision (64-bit) format, was used in [8] to facilitate OCaml code generation for floating-point numbers. Flocq [14], a Coq library of floating-point arithmetic, defines a type with similar semantics inductively, rather than using a quotient construction.

Most floating-point operations can then be lifted [25] in a straightforward manner from `('e,'f) float` to `('e,'f) floatSingleNaN`. We have additionally defined various operations that are supported in SMT-LIB but that were not available in Lei Yu's model, such as conversion functions between floating-point numbers and bit vectors. Our model now covers all operations that are available in the SMT-LIB floating-point theory.

Some (rather subtle) semantic differences between our model and the SMT-LIB floating-point theory remain. In SMT-LIB, the result of certain operations, such as converting NaN or infinities to a real number, is unspecified. Isabelle/HOL does not support partial specifications; therefore, the result of these operations is defined[2] in our model. Technically, the Isabelle/HOL model is an implementation of the SMT-LIB specification. This does not affect the soundness of interpreting the model in SMT-LIB: any theorem provable under SMT-LIB semantics also holds for the Isabelle/HOL model.

An error in the remainder function `float_rem` as defined in Isabelle/HOL was discovered during implementation and has been patched: the remainder of a finite floating-point value x and $\pm\infty$ shall be x [26, §5.3.1].

4 Interpreting Isabelle/HOL Floating-Point Arithmetic in SMT-LIB

This section describes an interpreted translation of floating-point types and operations from Isabelle/HOL to SMT-LIB. Our translation extends a preexisting general translation [11] targeting SMT solvers that is part of Sledgehammer, which treats floating-point arithmetic as uninterpreted. It supports the formal

[2] For instance, in terms of a special constant called `undefined`.

model of IEEE floating-point arithmetic in Isabelle/HOL that was described in the previous section. We aim to be comprehensive but restrict attention to those floating-point concepts that are defined in both Isabelle/HOL and SMT-LIB.

4.1 SMT-LIB Logic

The first task of our translation module is to select an SMT-LIB logic within which the SMT solver is to reason when deciding the satisfiability of the formula. For performance reasons, it is generally a good idea to select a logic that is as specific as allowed by the contents and structure of the formula. However, `FP`, the logic for floating-point arithmetic, is too restrictive for many of Isabelle's proof obligations, which may freely combine floating-point operations with other types and constants. When translated, these will require support for symbols that are either free (uninterpreted) or defined in other SMT-LIB theories.

Sledgehammer's SMT integration relies on callback functions to analyze the proof obligation and determine the problem's logic. However, only one of these functions may select a logic. In the absence of a framework allowing for a more modular approach (e.g., incrementally generalizing the logic as little as necessary, based on the types and constants that appear in the proof obligation), we need to select a logic that covers all operations that appear in the proof obligation. To achieve this, whenever a supported floating-point type is detected in the formula to be translated, our callback function returns the (pseudo-)logic `ALL`. Available since version 2.5 of the SMT-LIB standard, this provides a convenient way to select the most general logic that the respective SMT solver supports.

4.2 Types

Both Isabelle/HOL and SMT-LIB define binary floating-point formats of arbitrary width of the exponent and fraction fields. In Isabelle/HOL, `(m,n) floatSingleNaN` is the type of floating-point numbers with an exponent field of width `m` and a fraction field of width `n` (and thus with precision `n+1`). In SMT-LIB, the hidden bit of the significand (the bit preceding the fraction) is included in the format specification, making `(_ FloatingPoint m n+1)` the corresponding sort. The SMT-LIB sorts are only defined for formats with `m` $>$ 1 and `n` $>$ 0, whereas `m` and `n` are merely required to be positive in Isabelle/HOL. Thus, any type `(1,n) floatSingleNaN` lacks a corresponding sort in SMT-LIB, and is left uninterpreted by the translation.

In Isabelle/HOL, all floating-point formats `(m,n) floatSingleNaN` are instances of a polymorphic type `('e,'f) floatSingleNaN`. Here, `'e` and `'f` are type variables that may be instantiated with concrete (type) arguments, or left uninstantiated to express generic properties that hold for all floating-point formats. Due to the current lack of support for polymorphism in SMT-LIB, `(m,n) floatSingleNaN` is interpreted only when `m` and `n` are (type) arguments encoding fixed numeric values; polymorphic types are left uninterpreted.

In addition to the types for floating-point formats, Isabelle/HOL defines an enumerated type `roundmode` for the rounding modes used by the arithmetic

operations. SMT-LIB provides a corresponding type; `roundmode` is interpreted as `RoundingMode` in SMT-LIB.

4.3 Constants

For the sake of brevity, we focus here on some of the more interesting aspects of the translation of constants. (In HOL, constants are not limited to arity 0, but may have a function type.) An exhaustive enumeration of the mapping is provided in Table 1.

Polymorphism. The issue regarding polymorphism, described in the previous section, affects the translation of constants as well. A constant can only be interpreted if its type is not polymorphic. Since Isabelle's automatic type inference assigns constants the most general type possible with respect to the context, variables and constants with a floating-point type will in many cases need to be attached with explicit type constraints in order to trigger the interpretation.

Direct Correspondence. For many floating-point related constants in Isabelle, there is a direct semantic-preserving mapping to a function in SMT-LIB. Among these we find, e.g., the rounding modes and comparison operations together with many arithmetic operations and classification predicates. The translation of these does not involve much more than simply replacing their name with the corresponding identifier in SMT-LIB.

Format Parameter Extraction. A few SMT-LIB functions targeted by our translation are technically elements of an infinite family of functions generated by an index over all floating-point formats. This holds, e.g., for the conversion operation from reals to floating-points, and for the (nullary) functions denoting the special floating-point values ± 0, $\pm\infty$ and NaN. Their behavior depends on the result sort, which is not necessarily derivable from context and must be indicated explicitly in SMT-LIB. In these cases, we extract the type arguments of the (result) type of the constant to be translated, and add them explicitly as arguments to the corresponding function symbol in SMT-LIB. For instance, the Isabelle/HOL function `round` of type `roundmode` ⇒ `real` ⇒ `('e,'f) floatSingleNaN`, which converts a real number into a floating-point number (rounding as necessary), is interpreted as `(_ to_fp m n+1)` whenever its result type is of the form `(m,n) floatSingleNaN`, where `m` and `n` encode fixed numeric values.

Term Translation. Isabelle/HOL supports the definition of advanced concepts on top of the types and constants that are provided by the model of floating-point arithmetic. Our translation does not interpret such derived concepts directly. Instead, these can be handled by unfolding their definitions in Isabelle when desired, or by relying on Sledgehammer's relevance filter, which can make their definitions and other relevant facts available to external provers automatically.

Table 1. Types and constants in Isabelle/HOL covered by the translation, together with sorts and functions in SMT-LIB. m > 1 and n > 0 indicate the floating-point format. Square brackets denote syntactic sugar, which is also interpreted.

	Isabelle/HOL	SMT-LIB
Floating-point type	(m,n) floatSingleNaN	(_ FloatingPoint m n+1)
Rounding mode type	roundmode	RoundingMode
Bit-vector type	m word	(_ BitVec m)
Rounding mode	roundNearestTiesToEven	RNE
Rounding mode	roundNearestTiesToAway	RNA
Rounding mode	roundTowardPositive	RTP
Rounding mode	roundTowardNegative	RTN
Rounding mode	roundTowardZero	RTZ
Value construction	fp	fp
Positive infinity	plus_infinity [∞]	(_ +oo m n+1)
Negative infinity	minus_infinity	(_ -oo m n+1)
Positive zero	zero_class.zero [0]	(_ +zero m n+1)
Negative zero	minus_zero	(_ -zero m n+1)
Not-a-number	NaN	(_ NaN m n+1)
Absolute value	abs_class.abs [¦ ¦]	fp.abs
Negation	uminus_class.uminus [-]	fp.neg
Addition	fadd	fp.add
Subtraction	fsub	fp.sub
Multiplication	fmul	fp.mul
Division	fdiv	fp.div
Fused multiply-add	fmul_add	fp.fma
Square root	fsqrt	fp.sqrt
Remainder	float_rem	fp.rem
Integral rounding	fintrnd	fp.roundToIntegral
Less or equal	fle	fp.leq
Less than	flt	fp.lt
Greater or equal	fge	fp.geq
Greater than	fgt	fp.gt
IEEE equality	feq	fp.eq
Normal?	is_normal	fp.isNormal
Subnormal?	is_subnormal	fp.isSubnormal
Zero?	is_zero	fp.isZero
Infinity?	is_infinity	fp.isInfinite
NaN?	is_nan	fp.isNaN
Negative?	is_negative	fp.isNegative
Positive?	is_positive	fp.isPositive
To real	valof	fp.to_real
To unsigned word	unsigned_word_of_float	fp.to_ubv
To signed word	signed_word_of_float	fp.to_sbv
From IEEE word	float_of_IEEE754_word	(_ to_fp m n+1)
From real	round	(_ to_fp m n+1)
From float	float_of_float	(_ to_fp m n+1)
From signed word	float_of_signed_word	(_ to_fp m n+1)
From unsigned word	float_of_unsigned_word	(_ to_fp_unsigned m n+1)

5 Evaluation

To investigate the difference in the performance of Sledgehammer brought on by the interpreted translation, and to get a clear overview of the comparative performance of the SMT solvers, we conducted an experimental evaluation on a set of proof obligations that involve floating-point operations. Freely available Isabelle formalizations of floating-point properties are scarce; only a few properties are included with the formal IEEE model in the Archive of Formal Proofs. We complemented these with our own formalizations of floating-point properties taken from the IEEE 754 standard and the *Handbook of Floating-point Arithmetic* [32], resulting in a set of 124 formulas. The formulas in the evaluation set exhibit difficulties ranging from nearly trivial to levels on par with Sterbenz's lemma [42].

All formulas in the evaluation set are polymorphic over a single floating-point type `('e,'f) floatSingleNaN`. This type was instantiated to different fixed-size floating-point formats: half (16-bit), single (32-bit), double (64-bit), and quadruple (128-bit) precision formats, as specified by IEEE 754. The interpreted translation was evaluated on each of these fixed-size formats. For comparison, the abstract (uninterpreted) translation that was previously employed by Sledgehammer was additionally evaluated on the original (polymorphic) evaluation set. This gives rise to nine different models—technically, Isabelle theories with different type annotations—for measuring Sledgehammer's performance on the evaluation set, defined for $x \in \{\texttt{(5,10)}, \texttt{(8,23)}, \texttt{(11,52)}, \texttt{(15,112)}\}$ as:

- $\mathcal{I}_x$: interpretation is enabled and all floating-points are of type x `floatSingleNaN`.
- $\mathcal{U}_x$: interpretation is disabled and all floating-points are of type x `floatSingleNaN`.
- $\mathcal{U}_{\text{poly}}$: interpretation is disabled and all floating-points are of polymorphic type `('e,'f) floatSingleNaN`.

We used the Mirabelle [17] tool with default settings—including a 30 s time limit per formula—to apply Sledgehammer to each proof obligation. The default external provers invoked by Sledgehammer in Isabelle2022 are the ATPs E (version 2.6-1), SPASS (version 3.8ds-2), Vampire (version 4.6) and Zipperposition (version 2.1-1), along with the SMT solvers CVC4 (version 1.8), veriT (version 2021.06.2-rmx), and Z3 (version 4.4.0_4.4.1). Since the floating-point solver in this version of Z3 suffers from a soundness bug, we evaluated Z3 version 4.12.2 instead. We did not evaluate newer versions of the other solvers, such as cvc5 [3], as they are not yet integrated with Isabelle.

Out of the three SMT solvers, only CVC4 and Z3 support the floating-point theory of SMT-LIB. For each of the nine models, we evaluated four different prover configurations: CVC4 only, Z3 only, CVC4+Z3, and Sledgehammer's default prover configuration, which includes all of the ATPs and SMT solvers listed above. For the $\mathcal{I}_x$ models, where interpretation is enabled, the default prover configuration uses both interpreted and uninterpreted translations

(depending on the prover). For CVC4, we enabled its experimental floating-point solver (option `--fp-exp`) to obtain support for floating-point formats beyond single and double precision.

Sledgehammer's relevance filter had access to a large collection of theorems from the Isabelle/HOL library, including the definitions of all types and operations, and (for later formulas in the evaluation set) to all formulas that were evaluated earlier. This mimics realistic use in interactive proof, where users can rely on proven statements and employ them as lemmas in subsequent proofs. To avoid later runs being affected by earlier runs, the status of the machine learning selection of facts (stored in the Isabelle configuration file `mash_state`) was reset before each Mirabelle run.

The experiments were conducted under Debian GNU/Linux 6.1.0-10-amd64, running on an i9-9980HK CPU at 2.4 GHz with 16 processor threads and 32 GB of main memory.

5.1 Results

Table 2 shows Sledgehammer's success rates for the four different prover configurations when run on the evaluation set in the models described above. For convenience, the four fixed formats are abbreviated by their total bit length (16, 32, 64, and 128, respectively) in the model name. Sledgehammer succeeds when at least one of the external provers reports that it found a proof within the time limit of 30 s.

Table 2. Sledgehammer's success rates for the four prover configurations on proof goals from the evaluation set, by model.

	$\mathcal{U}_{16}$	$\mathcal{I}_{16}$	$\mathcal{U}_{32}$	$\mathcal{I}_{32}$	$\mathcal{U}_{64}$	$\mathcal{I}_{64}$	$\mathcal{U}_{128}$	$\mathcal{I}_{128}$	$\mathcal{U}_{\text{poly}}$
CVC4	41%	94%	57%	91%	35%	90%	58%	**89**%	54%
Z3	39%	86%	56%	85%	35%	84%	56%	77%	**58**%
CVC4+Z3	41%	**95**%	58%	**91**%	36%	90%	58%	**89**%	57%
Default (all)	41%	94%	60%	91%	37%	**91**%	60%	88%	56%

In this case, Sledgehammer attempts to reconstruct the external proof in Isabelle using a collection of automated proof methods (as discussed in Sect. 2.1). The success rates for this process, again as a percentage of the total number (124) of proof obligations, are shown in Table 3.

For each floating-point format (and also for the polymorphic model), the largest success rate across prover configurations, with or without interpretation enabled, is indicated in boldface.

Table 3. Success rates of proof reconstruction for the four prover configurations on proof goals from the evaluation set, by model.

	$\mathcal{U}_{16}$	$\mathcal{I}_{16}$	$\mathcal{U}_{32}$	$\mathcal{I}_{32}$	$\mathcal{U}_{64}$	$\mathcal{I}_{64}$	$\mathcal{U}_{128}$	$\mathcal{I}_{128}$	$\mathcal{U}_{\text{poly}}$
CVC4	**41%**	5%	55%	5%	35%	5%	54%	5%	54%
Z3	39%	4%	54%	4%	35%	4%	53%	4%	**58%**
CVC4+Z3	**41%**	5%	55%	5%	36%	5%	56%	5%	57%
Default (all)	40%	7%	**58%**	7%	**37%**	7%	**57%**	7%	54%

5.2 Discussion

Based on the results of our evaluation, we put forward the following observations:

1. *An interpreted translation increases Sledgehammer's success rate for all prover configurations and fixed-size floating point formats.* With an uninterpreted translation, success rates vary between 35% and 58%. This increases to between 77% and 95% with an interpreted translation. Across the board, the interpreted translation performs significantly better than the uninterpreted translation.
2. *The increase in Sledgehammer's success rate is most pronounced for the half (16-bit) and double (64-bit) precision formats.* The uninterpreted translation performs worse for these two formats (with success rates of 35% to 41%) than for single and quadruple precision. In contrast, the interpreted translation consistently yields high success rates (of 89% to 95% in the best solver configuration) regardless of the format's precision.
3. *Sledgehammer's success rate on the polymorphic model is generally comparable to, and in some cases better than, its success rate for fixed-size formats with an uninterpreted translation.* When the external provers cannot take advantage of their decision procedures for fixed-size floating-point arithmetic, reasoning about fixed-size properties is no easier for them than reasoning about polymorphic properties. (Indeed, depending on the additional facts chosen by Sledgehammer's relevance filter, it may well be harder.) This changes when interpretation is enabled.
4. *CVC4 outperforms Z3 on most models.* This is true both with and without interpretation enabled. The only exception is the polymorphic model, where Z3 performs slightly better than CVC4. Using all available provers typically results in (only) slightly higher success rates than using CVC4 alone, but can also lead to slightly lower success rates (mainly because of non-determinism in Sledgehammer's behavior).
5. *With interpretation disabled, proof reconstruction success rates are often close to Sledgehammer's success rates.* In other words, proof reconstruction in the uninterpreted models succeeds on the vast majority of proofs found by external provers. This is a testament to the power of Isabelle's built-in proof methods (in particular, `metis`), which provide strong automation for first-order reasoning.

6. *Interpretation leads to (much) lower proof reconstruction rates for all prover configurations and fixed-size floating point formats.* Although interpretation allows external provers to find more proofs, these proofs are rarely successfully reconstructed in Isabelle. This is to be expected: Isabelle currently does not offer built-in automated proof procedures for floating-point reasoning that could be used to reconstruct such proofs.

Many formulas from the evaluation set were previously proven with 10–20 lines of interactively developed Isabelle proof script, and can now (after interpretation) be proven completely automatically by CVC4 or Z3. The interpreted translation can save significant amounts of human labor in formal proof developments that involve floating-point arithmetic. However, due to the lower proof reconstruction rate, interpretation of floating-point arithmetic is currently primarily of interest to users who are willing to accept CVC4 and Z3 as oracles (cf. Sect. 2.1).

6 Related Work

The practice of employing automatic provers as back-ends in interactive theorem provers is not unique to Isabelle. Generic proof-delegation tools similar to Sledgehammer have also been developed for other proof assistants, e.g., Miz𝔸ℝ [43] for Mizar [2], and HOL(y)Hammer [27] for HOL Light [22] and HOL4 [41]. There are also proof-delegation tools aimed specifically toward SMT solvers, e.g., Smtlink [37] for ACL2 [28] and SMTCoq [1] for Coq [10].

Single integrations of SMT solvers have perhaps been more common than these larger-scale tools. The interactive theorem prover PVS [35] is tightly connected with the SMT solver Yices [18] (and its predecessor ICS), which has been available as a decision procedure for a long time. An oracle integration of Yices in Isabelle by Erkök and Matthews [20] makes use of its dedicated decision procedures, but refrains from translating into SMT-LIB, and instead targets the native input format of Yices due to its expressiveness. Weber [44] proposes a similar oracle integration of Yices into HOL4, but extends it with support for additional SMT solvers via the SMT-LIB format. This integration has since been supplemented with proof reconstruction and become part of HOL(y)Hammer [13].

The work presented here is based on the original integration of SMT solvers in Isabelle's Sledgehammer by Blanchette et al. [11]. It is dependent on various aspects of their translation into SMT-LIB, including the interpretation of bit-vector types and constants. In this sense, it also bears resemblance to how SMTCoq was recently extended with dedicated support for the theory of bit vectors [19].

Formalizations of IEEE 754 floating-point arithmetic are readily available in interactive proof assistants, e.g., in HOL Light [23], ACL2 [39], and Coq [14], and have been used extensively to verify floating-point related properties. However, to the best of our knowledge, no integration of SMT solvers in interactive proof assistants takes advantage of the dedicated decision procedures for floating-point arithmetic available in the former.

Superficially, the work perhaps most similar to ours is a Why3 [12] formalization of floating-point arithmetic and its mapping to the SMT-LIB floating-point theory [21]. Why3, however, is not a prover itself, but a stand-alone proof-delegation tool relying completely on external provers. Thus greater automation in interactive proof assistants is not a shared objective.

7 Conclusions

In the years since its introduction in Isabelle, Sledgehammer has seen a number of improvements. In varying degree, they have each gradually brought us closer to the ultimate goal of powerful proof automation in interactive proof assistants. By defining a formal model of floating-point arithmetic in Isabelle/HOL that implements SMT-LIB semantics, and by enhancing the translation from Isabelle to SMT-LIB with an interpretation of floating-point types and constants, we have taken another step in this direction. Sledgehammer enjoys a significant increase in success rates (before proof reconstruction) for proof obligations that involve floating-point arithmetic.

Many proof obligations that were previously out of reach for any automated prover can now be solved automatically. For users who are willing to trust the external SMT solvers, enhancing Sledgehammer's translation with a floating-point interpretation increases proof automation and reduces the manual effort required to construct proofs in this important application domain.

Our translation does not require formulas to be fully interpretable in the SMT-LIB floating-point theory. The SMT solvers are instructed to reason in a more general logic, where interpreted and uninterpreted sorts and functions can be combined freely.

There are two notable limitations, which we propose to address through future work. First, the interpretation of floating-point arithmetic is restricted to fixed-size formats. In many situations, this is not a severe limitation—fixed-size reasoning is sufficient, for instance, when one wants to verify a specific hardware architecture, or a software implementation that uses a specific floating-point type such as `binary64`. However, floating-point properties that hold for all formats are most naturally stated polymorphically in Isabelle/HOL. Such properties cannot be interpreted in the floating-point theory of SMT-LIB, which (in its current version 2.6) lacks support for polymorphism: although it offers a type `(_ FloatingPoint m n)` for any sufficiently large `m` and `n`, it does not offer a polymorphic type `(_ FloatingPoint` m n`)` where m and n are variables that may be instantiated.

Supporting polymorphism in SMT solvers is no small feat. Fortunately, there is ongoing work to obtain a tighter integration of automatic provers, including SMT solvers, with proof assistants. One of the means by which to achieve this is via support for higher-order logic in these provers [5]. Most likely, SMT-LIB 3—the next major update to SMT-LIB—will facilitate these changes by supporting polymorphism [4]. When such support becomes available in SMT solvers that support floating-point arithmetic, an interpreted translation can be employed

also for polymorphic floating-point properties. There has already been work on supporting parametric bit-vector formulas in SMT solvers by encoding them as formulas over non-linear integer arithmetic, uninterpreted functions, and universal quantifiers (the UFNIA logic in SMT-LIB) [33]. This approach could in principle be extended to floating-point numbers.

Second, interpretation of floating-point arithmetic allows SMT solvers to find more proofs, but reduces proof reconstruction rates in Isabelle. There is a mismatch between the reasoning capabilities of SMT solvers that support floating-point arithmetic and Isabelle's built-in automated proof procedures, which are used to reconstruct proofs. The latter currently do not offer dedicated support for floating-point reasoning, but need to rely on explicit lemmas to reason about concepts for which the SMT solver, when interpretation is enabled, can employ specialized decision procedures. Users may opt to bypass proof reconstruction and use external SMT solvers as oracles; however, this reduces trust in the resulting theorems, as errors in the SMT solver, in the translation from Isabelle/HOL to SMT-LIB, or in the Isabelle/HOL model of floating-point arithmetic could lead to unsound results. The approach preferred by the interactive theorem proving community is that of a skeptic [24]—external proofs should be reconstructed internally. If successful, this approach combines the speed of the SMT solver with the reliability of the proof assistant.

Efficient reconstruction of proofs has previously been achieved for other SMT-LIB logics [11], and is likely possible also for floating-point reasoning, through improving on the proof information provided by SMT solvers and translating theory-specific inferences. An automated proof procedure for floating-point arithmetic implemented on top of Isabelle's inference kernel would both facilitate the reconstruction of external proofs and increase the built-in automation for floating-point reasoning available in Isabelle/HOL. The implementation of such a proof procedure will require substantial work, but the evaluation results in this paper—in particular, the difference between Tables 2 and 3—clearly indicate that the effort would not be wasted.

Acknowledgments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to coq through proof witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12
2. Bancerek, G., et al.: Mizar: state-of-the-art and beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 261–279. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_17
3. 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

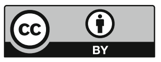
4. Barbosa, H., Blanchette, J.C., Cruanes, S., Ouraoui, D.E., Fontaine, P.: Language and proofs for higher-order SMT (work in progress). In: Dubois, C., Paleo, B.W. (eds.) Fifth Workshop on Proof eXchange for Theorem Proving - PxTP 2017. Electronic Proceedings in Theoretical Computer Science, vol. 262, pp. 15–22 (2017). https://doi.org/10.4204/EPTCS.262.3
5. Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 35–54. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_3
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). https://www.smt-lib.org/
7. 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
8. Basin, D., et al.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 432–453. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_25
9. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirovic, P.: Superposition for higher-order logic. J. Autom. Reason. **67**(1), 10 (2023). https://doi.org/10.1007/s10817-022-09649-9
10. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
11. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reason. **51**(1), 109–128 (2013). https://doi.org/10.1007/s10817-013-9278-5
12. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64 (2011)
13. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_14
14. Boldo, S., Melquiond, G.: Some formal tools for computer arithmetic: Flocq and Gappa. In: 28th IEEE Symposium on Computer Arithmetic, ARITH 2021, Lyngby, Denmark, 14–16 June 2021, pp. 111–114. IEEE (2021). https://doi.org/10.1109/ARITH51176.2021.00031
15. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
16. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: 22nd IEEE Symposium on Computer Arithmetic - ARITH 2015, pp. 160–167. IEEE (2015). https://doi.org/10.1109/ARITH.2015.26

17. Desharnais, M., Vukmirovic, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, 7–10 August 2022, Haifa, Israel. LIPIcs, vol. 237, pp. 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.8
18. 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
19. Ekici, B., Katz, G., Keller, C., Mebsout, A., Reynolds, A.J., Tinelli, C.: Extending SMTCoq, a certified checker for SMT (extended abstract). In: Blanchette, J.C., Kaliszyk, C. (eds.) First International Workshop on Hammers for Type Theories - HaTT@IJCAR 2016. Electronic Proceedings in Theoretical Computer Science, vol. 210, pp. 21–29 (2016). https://doi.org/10.4204/EPTCS.210.5
20. Erkök, L., Matthews, J.: Using Yices as an automated solver in Isabelle/HOL. In: Rushby, J., Shankar, N. (eds.) AFM 2008: Third Workshop on Automated Formal Methods, pp. 3–13 (2008)
21. Fumex, C., Marché, C., Moy, Y.: Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria Saclay Ile de France (2017). https://hal.inria.fr/hal-01511183
22. Harrison, J.: HOL light: a tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031814
23. Harrison, J.: Floating-point verification using theorem proving. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006. LNCS, vol. 3965, pp. 211–242. Springer, Heidelberg (2006). https://doi.org/10.1007/11757283_8
24. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. J. Autom. Reason. **21**(3), 279–294 (1998). https://doi.org/10.1023/A:1006023127567
25. Huffman, B., Kunčar, O.: Lifting and transfer: a modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_9
26. IEEE standard for floating-point arithmetic. IEEE STD 754-2019 (Revision of IEEE 754-2008), pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
27. Kaliszyk, C., Urban, J.: `HOL(y)Hammer:` online ATP service for `HOL Light`. Math. Comput. Sci. **9**(1), 5–22 (2014). https://doi.org/10.1007/s11786-014-0182-0
28. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on Common Lisp. IEEE Trans. Software Eng. **23**(4), 203–213 (1997). https://doi.org/10.1109/32.588534
29. Klein, G., et al.: seL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574
30. 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
31. 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
32. Muller, J.M., et al.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010). https://doi.org/10.1007/978-0-8176-4705-6

33. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. J. Autom. Reason. **65**(7), 1001–1025 (2021). https://doi.org/10.1007/s10817-021-09598-9. http://www.cs.stanford.edu/barrett/pubs/NPR+21c.pdf
34. 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
35. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
36. 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. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2010). https://easychair.org/publications/paper/wV
37. Peng, Y., Greenstreet, M.R.: Extending ACL2 with SMT solvers. In: Kaufmann, M., Rager, D.L. (eds.) Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications - ACL2 2015. Electronic Proceedings in Theoretical Computer Science, vol. 192, pp. 61–77 (2015). https://doi.org/10.4204/EPTCS.192.6
38. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. **15**(2–3), 91–110 (2002). http://content.iospress.com/articles/ai-communications/aic259
39. Russinoff, D.M.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS J. Comput. Math. **1**, 148–200 (1998). https://doi.org/10.1112/S1461157000000176
40. 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
41. 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
42. Sterbenz, P.H.: Floating-Point Computation. Prentice-Hall, Hoboken (1974)
43. Urban, J., Rudnicki, P., Sutcliffe, G.: ATP and presentation service for Mizar formalizations. J. Autom. Reason. **50**(2), 229–241 (2013). https://doi.org/10.1007/s10817-012-9269-y
44. Weber, T.: SMT solvers: new oracles for the HOL theorem prover. Int. J. Softw. Tools Technol. Transfer **13**(5), 419–429 (2011)
45. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 1965–2013. Elsevier and MIT Press (2001). https://doi.org/10.1016/b978-044450813-3/50029-1
46. Yu, L.: A formal model of IEEE floating point arithmetic. Archive of Formal Proofs (2013). http://isa-afp.org/entries/IEEE_Floating_Point.html. Formal proof development

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.



Learning Proof Transformations and Its Applications in Interactive Theorem Proving

Liao Zhang[1,2(✉)], Lasse Blaauwbroek[3], Cezary Kaliszyk[1,4], and Josef Urban[2]

[1] University of Innsbruck, Innsbruck, Austria
zhangliao714@gmail.com
[2] Czech Technical University in Prague, Prague, Czech Republic
[3] Institut des Hautes Etudes Scientifiques Paris, Paris, France
[4] International Neurodegenerative Disorders Research Center, Prague, Czech Republic

Abstract. Interactive theorem provers are today increasingly used to certify mathematical theories. To formally prove a theorem, reasoning procedures called tactics are invoked successively on the proof states starting with the initial theorem statement, transforming them into subsequent intermediate goals, and ultimately discharging all proof obligations. In this work, we develop and experimentally evaluate approaches that predict the most likely tactics that will achieve particular desired transformations of proof states. First, we design several characterizations to efficiently capture the semantics of the proof transformations. Then we use them to create large datasets on which we train state-of-the-art random forests and language models. The trained models are evaluated experimentally, and we show that our best model is able to guess the right tactic for a given proof transformation in 74% of the cases. Finally, we use the trained methods in two applications: proof shortening and tactic suggesting. To the best of our knowledge, this is the first time that tactic synthesis is trained on proof transformations and assists interactive theorem proving in these ways.

Keywords: Interactive theorem proving · Machine learning · Neural networks

1 Introduction

Interactive theorem provers (ITPs) [15] are sophisticated systems used for constructing machine-verified proofs. Various proof assistants, such as HOL4 [31], HOL Light [14], Lean [23], Isabelle/HOL [24], and Mizar [3], are used by formalizers. Coq [33] is one of the most popular proof assistant systems. Coq formalizers invoke reasoning procedures called *tactics* that transform proof states into simpler proof states, eventually discharging all proof obligations and thus proving the initial proof state.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 236–254, 2023.
https://doi.org/10.1007/978-3-031-43369-6_13

```
Theorem rev_length : ∀ l : list nat, length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - reflexivity.
  - simpl. rewrite → app_length. simpl. rewrite → IHl'.
    rewrite add_comm. reflexivity.
Qed.
```

Fig. 1. A formal Coq proof, showing the equality property of the lengths of a list and its reverse

To give a simple example, we show a Coq proof of the equality of the lengths of a list and its reverse (Fig. 1). To complete the proof, one can perform induction on the list `l` (with the help of the tactic `induction l as [| n l' IHl']`), splitting the proof state into a case where `l` is empty and a case where `l` is nonempty. In the first case, the goal reduces to `length (rev []) = length []`, which is easily discharged using simple computation. In the second case, we obtain the induction hypothesis `IHl'` that states `length (rev l') = length l'` and need to prove that the equation still holds when the original list has a natural number `n` prepended to it. After some simplification, we transform the length of the concatenation of two lists into the summation of their individual lengths. Then, with the help of the induction hypothesis, we simplify the goal. Finally, we rewrite the goal by the commutative property of addition and obtain a simple equation to prove.

A Coq proof state consists of a list of hypotheses and a goal that needs to be proven. Given a proof state before the tactic application, the tactic may either transform the *before state* to several *after states* or finish the proof. The *semantic* of a tactic is captured by the (usually infinite) set of proof state transformations that can potentially be generated by that tactic. In this work, we approximate that infinite set with a finite dataset of transformations that occur in real proofs written by Coq users. We then use machine learning models to gain an understanding of tactics using their approximated semantics.

As an example, Fig. 2 presents the before and after states of the tactic `rewrite add_comm` at its position in Fig. 1. In this particular case, the hypotheses remain unchanged, but in the goal, the two sides of the addition are swapped.

```
n : nat
l' : list nat
IHl' : length (rev l') = length l'
____________________________________(1/1)
length l' + 1 = S (length l')
```

(a) Before state

```
n : nat
l' : list nat
IHl' : length (rev l') = length l'
____________________________________(1/1)
1 + length l' = S (length l')
```

(b) After state

Fig. 2. The before and after states of `rewrite add_comm` in Fig. 1, with hypotheses above the dashed line and the required goal below it.

In this paper, we consider the machine learning task of predicting a tactic capable of generating a given proof state transformation and investigate the applications of this task. Formally, given a before state ps and n after states $\{ps'\}_{1..n}$, we attempt to predict a tactic t that transforms ps to $\{ps''\}_{1..n}$ such that ps'_i is equal to ps''_i modulo α-equivalence for every i.

1.1 Motivation

Tactic prediction methods have so far relied solely on before states, typically to guide automated tactical proof search in systems like Tactician [6]. We are interested in synthesizing tactics based both on the before and after states for a number of reasons.

First, there are multiple interesting applications of this task. For example, formalizers may want to arrive at a particular proof state, given a particular initial proof state. Or, given particular before and after states that were generated with a sequence of tactics, we may want to find a *single* tactic capturing the transformation, thus shortening and simplifying the proof, and teaching the formalizer how to use the available tactics.

Second, our work is the first step to designing a novel human-like proof search strategy. When mathematicians write pencil-and-pen proofs, they often first imagine some intermediate goals and then sequentially fill in the gaps. This provides another motivation: our trained predictors can recommend the tactics that will bridge the gaps between such intermediate human-designed proof goals.

Third, the task can be of particular importance for the ITPs which support constructing proofs in a declarative proof style, such as Isabelle, Mizar, and Lean. In declarative-style proofs often the after states are specified by the user manually. A large formal library, Mizar Mathematical Library [2], is developed in a declarative style. The Isabelle Archive of Formal Proofs (one of the most developed libraries today) is also predominantly written in a declarative style. Our approach can be directly applied to predict tactics able to fill the gap between two subsequent declarative statements.

Finally, the learned tactic embeddings could be used to perform MuZero-style [30] reinforcement learning, which means obtaining the after states by combining the embeddings of the before states and of the tactics without actually running the ITP. This could be particularly useful when some tactic applications require large computational resources.

1.2 Contributions

The main contributions of our paper can be summarized as follows.

1. To our best knowledge, we are the first to predict tactics based on the transformation they make between before and after states.
2. In Sect. 2, to capture the semantics of tactics, we design three characterizations: feature difference, anti-unification, and tree difference.

3. In Sect. 4, we conduct experiments to verify the strengths of our characterizations with a random forests classifier and the GPT-2 language model.
4. In Sect. 5, we propose and evaluate two applications of the task, namely tactic suggestion and proof shortening.

Besides the above-mentioned contributions, Sect. 3 introduces the preliminaries of the learning technology used in this paper. We discuss two related research fields in Sect. 6. The conclusions and future work are presented in Sect. 7.

2 Proof State Characterizations

To train the machine learning models, we need to provide characterizations of the before and after states. Apart from directly using the unprocessed textual representation of proof states, we design three characterizations: feature difference, anti-unification, and tree difference.

2.1 Feature Difference

To characterize the proof states, we start with the features used by [42]. In that work, the features were used to apply machine learning to predict tactics for proof states. For example, `GOAL-$l'` and `HYPS-Coq.Lists.List.rev-$l'` are two features extracted from the before state in Fig. 2. The prefixes `GOAL` and `HYPS` denote whether a feature belongs to the goal or the hypotheses. The symbol `$l'` denotes a node that occurs in the abstract syntax tree (AST) of the proof state. The prefix `$` means that `l'` denotes a named variable. We subsequently consider the nodes connected in the AST. For example, the feature `Coq.Lists.List.rev-$l'` means that the identifier of the reversion operation of a list and the list `l'` are connected in the AST.

For the current work, we additionally consider feature difference. From the before state ps and after states $\{ps'\}_{1..n}$, we extract features f and $\{f'\}_{1..n}$, respectively using the procedure discussed above. We define f' as the union of $\{f'\}_{1..n}$. By set difference, we compute the *disappeared features* $f - f'$ and the *appearing features* $f' - f$. The disappeared features and appearing features are together used as feature difference characterization of the tactic.

2.2 Anti-unification

Anti-unification, first proposed by Plotkin [27] and Reynolds [29], aims to calculate generalizations of the given objects. Since Coq is based on the Calculus of Inductive Constructions (CIC) [25], an appropriate anti-unification algorithm for Coq should be higher-order. However, higher-order anti-unification is undecidable [26]. Therefore, we first convert Coq terms to first-order terms so that we can execute a decidable and efficient first-order anti-unification algorithm.

To encode Coq terms into first-order logic, we transform them recursively following the AST. First-order applications and constants are encoded directly,

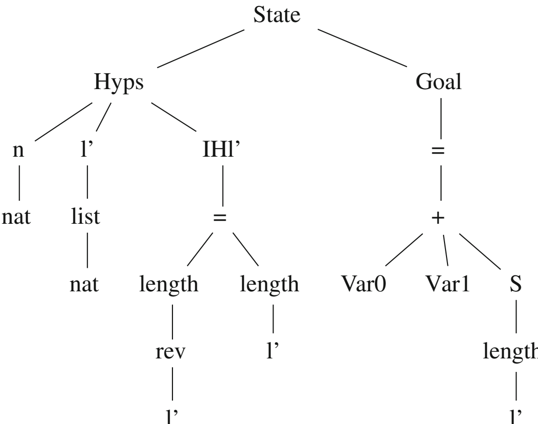


Fig. 3. The least general generalization of the before and after states in Fig. 2

other applications use the apply functor `app` and all other cases use special first-order functions (e.g., a dependent product is encoded as a first-order function `prod`). The goal of the before state in Fig. 2 will be converted to the first-order term $=(+(length(l'), S(O)), S(length(l')))$. The non-leaves $=, +, length, S$ denote function symbols. The leaves l' and O denote constants.

Terms in first-order anti-unification are defined as $t ::= x \mid a \mid f(t_1, ..., t_n)$ where x is a variable, a is a constant, f is an n-ary function symbol, and t is a term. In this paper, letters s, t, u denote terms, letters f, g, h denote function symbols, letters a, b denote constants, and letters x, y denote variables. *Substitutions* map variables to terms and are usually written in the form of sets. We can represent a substitution σ as a set $\{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$ where $\sigma(x)$ is the term mapped by x. The application of a substitution σ to a term t is represented as $t\sigma$. If t is a variable, then $t\sigma = \sigma(t)$. If $t = f(t_1, ..., t_n)$, then $t\sigma = f(t_1\sigma, ..., t_n\sigma)$. A term u is called a *generalization* of a term t if there exists a substitution σ such that $u\sigma = t$. For instance, the term $f(g(x), y)$ is a generalization of the term $f(g(a), h(a, b))$. The substitution σ is $\{x \mapsto a, y \mapsto h(a, b)\}$ such that $f(g(x), y)\sigma = f(g(a), h(a, b))$.

Anti-unification aims to obtain the *least general generalization (lgg)* of two terms s and t. A term u is called a generalization of s and t if there exist substitutions σ_1 and σ_2 such that $u\sigma_1 = s \wedge u\sigma_2 = t$. A generalization u' of s and t is called the lgg if, for any generalization u of s and t, there is a substitution σ, such that $u'\sigma = u$. Assuming ϕ is a bijective function from a pair of terms to a variable, given two terms s and t, the anti-unification algorithm AU calculates the lgg using the two rules below.

- $AU(s, t) = f(AU(s_1, t_1), ..., AU(s_n, t_n))$ if $s = f(s_1, ..., s_n)$, $t = f(t_1, ..., t_n)$
- $AU(s, t) = \phi(s, t)$ if the preceding rule does not match.

Figure 3 presents the lgg of the before and after states considered in Fig. 2. Compared to the before state, most of the nodes in the lgg remain the same.

The differences stay in the left side of the equality in the goal: `length l'` is substituted with `Var0`, and the natural number `1` is substituted with `Var1`. We need to apply the substitutions $\{var_0 \mapsto length\ l', var_1 \mapsto 1\}$ and $\{var_0 \mapsto 1, var_1 \mapsto length\ l'\}$ to the lgg to obtain the before and after states, respectively.

We compute the *lggs* of the goals and the hypotheses separately. We can directly anti-unify the goals of the before and after states. However, the number of hypotheses may be changed by the tactic application. For instance, the tactic `intros` introduces new hypotheses, while the tactic `clear H` removes the hypothesis `H`. Suppose we are anti-unifying the hypotheses $hyps(h_1, ..., h_n)$ and $hyps(h_1, ..., h_n, h_{n+1})$. The first rule of anti-unification immediately fails, and the second rule will generate a variable that corresponds to all hypotheses in the before state and all hypotheses in the after states. Therefore, anti-unifying all hypotheses together prevents us from developing a compact characterization. To calculate the lggs of hypotheses, we first match the hypotheses with the same names. Then, we compute an lgg on each pair. We refer to the hypotheses that are only in the before state and only in the after state as respectively *deleted hypotheses* and *inserted hypotheses*. Different from the pairwise hypotheses, we do not perform anti-unification on the deleted hypotheses and inserted hypotheses, and they remain unchanged.

We choose anti-unification because it can generate a more compact representation compared with directly utilizing the before and after states. Consider Fig. 2, we need a Coq string of the before state and another Coq string of the after state to characterize the transformation. Notice that many parts of the before state are unchanged after the tactic application. It is redundant to represent these unchanged parts twice in both the before and after states. However, anti-unification enables us to use a single lgg and the substitutions to characterize the transformation. The unchanged parts of the before and after states are shared in the lgg. Moreover, previous research has demonstrated that features based on generalization are very helpful for theorem proving [19].

2.3 Tree Difference

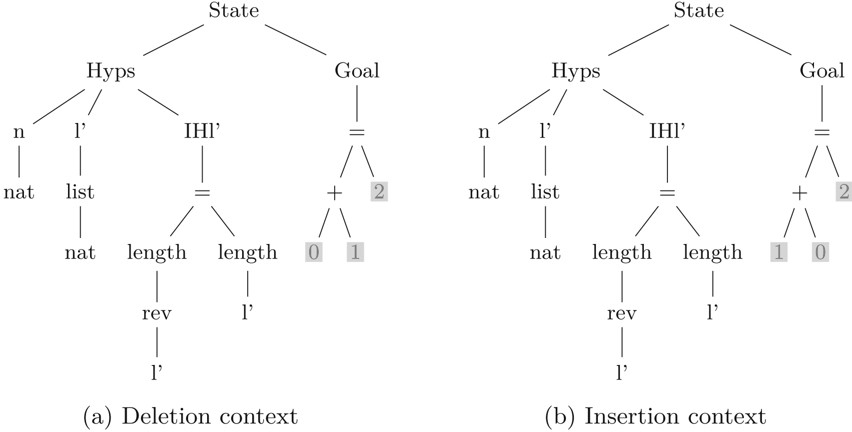
In addition to anti-unification, we propose a characterization based on a tree difference algorithm [21]. Compared to anti-unification, tree difference is better at generalizing the differences between the before and after states. Tree difference extends the standard Unix diff [16] algorithm by the capability to compute the differences according to the tree structures. Since proof states have tree structures, such tree differences can be used to characterize the transformations.

Take the before and after states in Fig. 2 for demonstration. First, for the hypotheses that are the same in the before and after states, we keep them unchanged. Therefore, the hypotheses `n`, `l'`, and `IHl'` remain the same.

The next step is to extract common subtrees from the original trees (except for the unchanged hypotheses) to obtain more compact characterizations. We focus on the ASTs of Coq terms. Assuming there is an oracle to judge whether the current subtree is a common subtree, we traverse a tree from the root. The calculation of the oracle is explained in the original paper [21]. If the current

subtree is a common subtree and not a leaf node, we substitute it with a hole. We do not substitute leaves with holes because, in practice, the substitutions of leaves lead to many unexpected holes. The same common subtrees should always be substituted with the same hole. The results of applying the substitutions to the before and after states are called the *deletion context* and the *insertion context*, respectively. After the substitutions, the deletion and insertion contexts are shown in Fig. 4.

Afterward, we calculate the *greatest common prefix (gcp)* of the deletion and insertion contexts and obtain a *patch*. According to the original algorithm, if the two trees have the same non-hole node, we keep the node unchanged and execute the algorithm on their children. Otherwise, we denote them as a *change*.



(a) Deletion context (b) Insertion context

Fig. 4. The deletion and insertion contexts of the before and after states in Fig. 2. `Hole0`, `Hole1`, and `Hole2` denote `length l'`, `1`, and `S(length l')`, respectively.

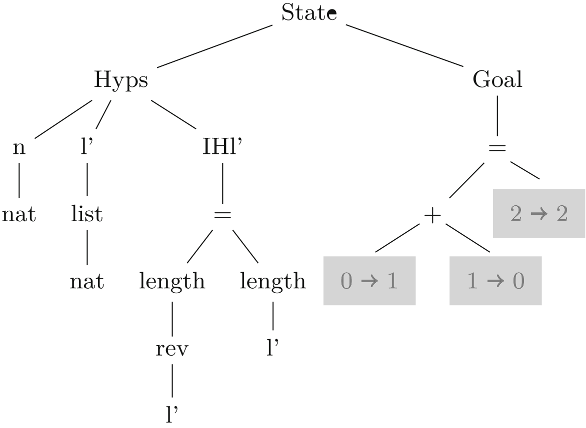


Fig. 5. The patch of the before and after states in Fig. 2

Similar to anti-unification, due to the deletion, insertion, and reordering of the hypotheses, we need to adjust the gcp algorithm for proof states. We match hypotheses by their names and obtain the deleted hypotheses, inserted hypotheses, and matched hypotheses as in Sect. 2.2. We only calculate gcps on the matched hypotheses. The deleted hypotheses and inserted hypotheses are represented as a change. Executing gcp on proof states returns a patch in the format of $state(hyps_patch, goal_patch)$ where $hyps_patch$ is constructed by $hyps(h_1, ..., h_n, change(del_hyps, ins_hyps))$. Each h_i is the patch of two matched hypotheses. Figure 5 depicts the patch of the before and after states in Fig. 2.

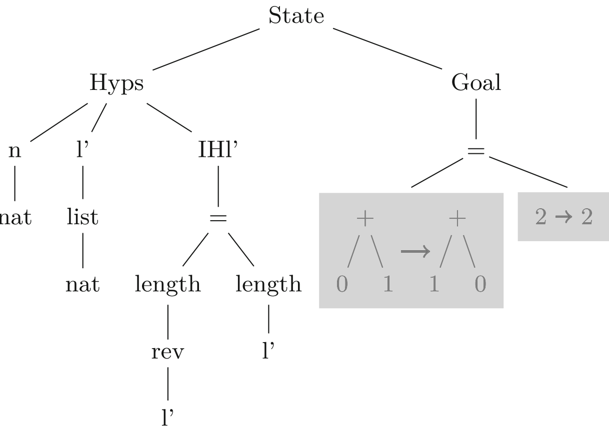


Fig. 6. The result of applying the closure function to the patch in Fig. 5

Subsequently, we need to calculate the *closure* of a patch. The intention is to confirm that every change is *closed*: the left and right sides contain the same holes. Notice that the patch in Fig. 5 contains two unclosed changes, `Change(Hole0, Hole1)` and `Change(Hole1, Hole0)`. The closure function will go to the subtree, whose root is the parent node of the unclosed change. Then, restore the subtree with the deletion and insertion contexts before we execute gcp on them. The procedure repeats until all changes are closed. Since the gcp function on proof states also returns a patch in a tree structure, we can run the closure function on it. If any patch of matched hypotheses h_i or $change(del_hyps, ins_hyps)$ are not closed, we restore the $hyps_patch$ with the original deletion and insertion contexts of the hypotheses. Then, if the $goal_patch$ or the deletion and insertion contexts of the hypotheses are not closed, we restore the patch of the proof states with the entire deletion and insertion contexts of the two proof states. Figure 6 depicts the patch after the execution of the closure function.

The final step is to replace the identical changes with their origin term. The original algorithm may cause identical changes, such as `Change(Hole2, Hole2)` in Fig. 6. Since we want a compact characterization, they are not necessary.

Tree difference is better at generalizing the differences compared to anti-unification. Take the example in Fig. 2 for instance. The lgg in Fig. 3 merely shows that the proof state changes in the position of the variables. The substitutions may be different if we execute `rewrite add_comm` on different proof states. However, in the patch generated by the tree difference in Fig. 6, the changes are generalized because we substitute common subterms with holes and will be the same even if we execute `rewrite add_comm` on different proof states.

2.4 Input Formats

During training, the language model receives the string `<Characterization> Tactic: <Tactic>` as input. `<Characterization>` has four variations:

- `Before:<Before State>`
- `Before:<Before State> After:[<After State>]`
- `Anti:[<Substs> <Delete_hyps> <Insert_hyps> <Lgg>]`
- `TreeDiff:[<Patch> <Hole>]`

A proof state is represented as a sequent `<Hyps> |- <Goal>`. The plain texts (like `Tactic:`) serve as prompts, while the placeholders (such as `<Before State>` and `<Tactic>`) are substituted according to the proof context. `[]` denotes a list. During prediction, the language model receives `<Characterization> Tactic:` as input and outputs the predicted tactics.

Random forests are fed discrete features as input. For feature difference, the disappeared features and appearing features are distinguished from each other (appearing features and disappeared features as introduced in Sect. 2.1). To utilize anti-unification, we convert the lgg and the terms in the substitution that should be used to obtain the before and after states to features in three disjoint spaces. For anti-unification, we also distinguish the features of deleted hypotheses and inserted hypotheses from other ones. For tree difference, we distinguish the gcp of the proof states, the origin and the destination of changes, and the common subterms into four spaces.

3 Learning Models

We consider two machine learning models for the task. The models will be compared experimentally in the next section.

The first model is a random forest classifier [7]. Random forests are based on decision trees. In decision trees, leaves represent labels (tactics in our case), and internal nodes correspond to features. A rule is a path from the root to a non-leaf. It represents the conjunction of all features on the path. A rule is determined by maximizing the *information gain* of examples. For instance, if we

have examples with labels $\{b, b, b, a, a\}$, we want to generate a rule that passes all examples with the label a to its left child and all examples with the label b to its right child. A forest makes predictions by voting based on a large number of decision trees. Random forests contain several sub-forests. Each sub-forest is built on a random subset of the entire dataset. We choose a random forest implementation that has previously been used to predict tactics for Coq [42].

The other used machine learning technique is the pre-trained language model GPT-2 [28]. GPT-2 is based on neural networks, which consist of many artificial neurons to learn from training data. The self-attention [35] technique is intensively applied in GPT-2 to differentially weigh every part of the input data. As a language model, GPT-2 predicts the probability distribution of the next word given a sequence of words as the input. GPT-2 is a pre-trained language model. The concept of pre-training imitates the learning process of humans. When humans encounter a new task, humans do not need to learn it from scratch. They will transfer and reuse their old knowledge to learn to solve it. Similarly, GPT-2 is pre-trained on a large natural language dataset BooksCorpus [43]. Afterward, GPT-2 can reuse the knowledge of natural language learned from pre-training to solve new tasks. To be adapted to a new task, we need to fine-tune GPT-2 on a relatively small dataset and slightly modify the weights learned from pre-training. We decide on GPT-2 because pre-trained language models have recently demonstrated outstanding achievements in natural language process (NLP) [8] and formal mathematics [34,39].

4 Experiments

We perform the experiments on the dataset extracted from the Coq standard library. The dataset consists of 158, 494 states extracted from 11, 372 lemmas. We randomly split the dataset into three subsets for training, validation, and testing in an 80-10-10% ratio. First, we use 100 trees by default and optimize the Gini Impurity [22]. Gini Impurity is a metric of the information gain. After the optimization, we set the Gini Impurity to its best value, try various numbers of trees and obtain the optimized number of trees. Finally, the best combination of Gini Impurity and the number of trees is determined for each characterization. The experiments with GPT-2 are based on the Hugging Face library [38]. In particular, we employ the smallest GPT-2. The hyper-parameters are: $eta = 3e - 4, num_beams = 3, batch_size = 32$. During training, we apply a linear schedule with the first 20% training steps for warm-up. The remaining parameters are left as their default values. At most 50 tokens are predicted for a single tactic. We truncate the input on the left side if it is longer than the maximal length limitation of GPT-2 (1024 tokens). Language models have length limitations for efficiency. The attention mechanism used by them causes a quadratic usage of memory as the length of tokens scales. Every model is trained for 25 epochs on an NVIDIA V100 GPU, and the snapshot with the highest accuracy on the validation dataset is selected for testing.

Table 1 depicts the results of our experiments. The accuracies of the combinations of before states with after states are significantly better than only relying

Table 1. Results on the test dataset, showing how often the prediction makes the same transformation as the tactic in the library. The transformations are considered modulo α-equivalence.

	random forests	GPT-2
before	43.23%	46.84%
before after	52.17%	67.45%
feature difference	59.34%	–
anti-unification	58.59%	71.74%
tree difference	58.98%	73.83%

on the before states in both random forests and GPT-2. Thus, we conclude that taking after states into consideration is very helpful to learn the semantics of tactics. The accuracies of GPT-2 are significantly higher than random forests, which confirms that the pre-trained language model is a more advanced machine learning technique compared to random forests. For random forests, all of the feature difference, anti-unification, and tree difference perform better than the unprocessed before and after states. This indicates that our characterizations can extract more precise features for random forests. We do not apply GPT-2 to feature differences, as it relies on natural language. In principle, it would be possible to give it feature differences directly as input, but as there are very few similarities between features and natural language it would be a serious disadvantage to the model. The knowledge grasped by pretraining is difficult to be used to understand features. Although feature difference is a little better than anti-unification and tree difference, their results are quite similar. The probable explanation is that random forests are not good at learning from sophisticated features. Random forests cannot learn meaningful knowledge from all three characterizations and almost only learn to make correct predictions for the simple tactics. Similarly, with GPT-2, anti-unification and tree difference provide more accurate predictions than the unprocessed before and after states. We suppose the explanation is that we are able to appropriately shorten the length of the input and also keep important information about the proof transformation. Appropriately shortening the input length is beneficial for GPT-2 because it has a maximal limitation on the number of input tokens. Table 2 compares the percentages of the inputs that are longer than the maximal length limitation. The statistics show that our implementation significantly reduces the probability that the input is over the maximal length limitation. Tree difference can provide more accurate predictions compared to anti-unification with both random forests and GPT-2. This may be attributed to that the generalization made by tree difference is easier to learn by machine learning models.

Table 2. The ratios of how many inputs exceed the maximal length limitation

	before	before after	anti-unification	tree difference
ratio	2.07%	7.96%	4.07%	3.90%

5 Applications

In this section, we propose two promising applications of the task. We only evaluate the most accurate of the methods proposed in the previous Sect. 4 (GPT-2) on the two tasks.

The first, more direct application, is making tactic suggestions. Given a before state, it is common for an ITP user to have an intuition of the intermediate proof states that are necessary to complete the proof. However, sometimes the user cannot guess the appropriate tactic needed to make the transformations. Using our model with the before state and the imagined intermediate states, the user can get a complete proposed proof as output. Hence, our model will predict the likely tactics to perform the transformations.

The other application is shortening existing Coq proofs. Specifically, for the transformation $ps_0 \Rightarrow_{t_0} ps_1 \Rightarrow_{t_1} ps_2 ... \Rightarrow_{t_n} ps_{n+1}$, where ps is a proof state and t is a tactic, we want to predict a tactic t' such that $ps_0 \Rightarrow_{t'} ps'$ where ps' and ps_{n+1} are equal under α-equivalence. Thus, we can replace the tactic sequence with a single tactic and decrease the length of the Coq proof. A restriction for this task is that because we are only interested in exploring shorter paths between proof states, ps_{n+1} should not be a finishing state.

Table 3. The first five tactics suggested by each characterization. The tactics displayed in bold result in the desired after states.

	before	before after	anti-unification	tree difference
1	trivial	**`rewrite <- minus_n_O`**	**`rewrite <- minus_n_O`**	**`rewrite sub_0_r`**
2	simpl	**`rewrite sub_0_r`**	**`rewrite Nat.sub_0_r`**	**`rewrite Nat.sub_0_r`**
3	**`rewrite <- minus_n_O`**	*rewrite*<− minus_n_0	simpl	simpl
4	*rewrite*<− plus_n_O	simpl	**`rewrite sub_0_r`**	*rewrite*<− sub_0_r
5	auto	*rewrite*<− sub_0_r	*rewrite*<− plus_n_O	apply sub_0_r

5.1 Tactic Suggestion

We view the experiments in Sect. 4 as the evaluation of tactic suggestions. The before and after states extracted from the Coq standard library are considered as the states that are presented in the Coq editor and those in users' minds, respectively. The results show that taking the after states into consideration, together with the more compact characterization, is essential for correctly suggesting tactics.

The following is an actual tactic suggestion question taken from the Coq Discourse Forum[1]. The question can be summarized as finding a tactic that transforms the following before state to the after state. The goal of the before state is to prove that the element indexed by $m - 0$ in a list equals the element indexed by m.

- Before state: `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth (m - 0) l 0 = nth m l 0`
- After state: `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth m l 0 = nth m l 0`

Table 3 shows the first five tactics predicted by each model. If we consider only the before state, we will obtain the correct prediction in the third place. However, the first two synthesized tactics using anti-unification, tree difference as well as unprocessed before and after states are appropriate. Besides the tactics displayed in bold, other tactics do not perform the expected transformation due to various reasons. Some tactics such as `trivial`, `simpl`, and `auto` do not change the proof state. The tactics `rewrite <- plus_n_O` and `apply sub_0_r` are not applicable and cause errors. The lemma `minus_n_0` used in `rewrite <- minus_n_0` does not exist in the Coq standard library. Although `rewrite <- sub_0_r` does not cause an error, it leads to an unexpected after state `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth (m - 0) l 0 = nth m l 0 - 0`. Since the operations executed by `trivial`, `simpl`, and `auto` are quite complicated and may depend on the context, we assume it is difficult for the model to comprehensively understand them. Their occurrences in the first five predictions may be mainly because they occur quite frequently in the training data. The results confirm that the combination of before and after states is beneficial for suitably suggesting tactics.

5.2 Shortening Proofs

The results presented in the previous Sect. 4 focused on decomposed tactics. This means compound tactic expressions that perform several steps at once have been decomposed into individual tactic invocations. We apply the technique that is developed by [5] to decompose the tactics. Here, we utilize the same models; however, we focus on the original human-written tactics and try to shorten these (shortening expanded tactics would be unfair). For all tactic sequences of lengths two and three in the training dataset, we input their before and after states into the model. In our experiment, we can only consider the states in the training dataset since our model is trained on all present tactics. Compared to the validation dataset and testing dataset, our model should be able to give better predictions on proof shortening for the training dataset. The amount of original tactics in the training dataset is 56,788. The model synthesizes 10 tactics for each sequence, and we execute them in Coq to verify that they perform the same transformation as the sequence modulo α-equivalence.

[1] https://coq.discourse.group/t/how-to-avoid-awkward-assertions/1153/2.

Table 4. The shortening ratios and amounts of redundant tactics with different characterizations and sequence lengths.

length		before	before after	anti-unification	tree difference
2	ratio	0.379%	0.824%	0.891%	0.833%
	number	215	468	506	473
3	ratio	0.039%	0.148%	0.151%	0.148%
	number	22	84	86	84

The results are presented in Table 4. We define the number of *redundant tactics* of $ps_0 \Rightarrow_{t_0} ps_1 \Rightarrow_{t_1} ps_2 ... \Rightarrow_{t_n} ps_{n+1}$ as n. The *shortening ratio* is defined as the number of all discovered redundant tactics divided by the total number of occurrences of tactics in the training dataset. In this section, our method only applies to a tactic sequence that, besides the last tactic, every intermediate tactic produces a single after state. While in Sect. 4, our experiments apply to tactic applications that may produce several after states. The reason is that it is difficult to calculate the number of redundant tactics if intermediate tactics produce several after states. The tactic sequence will become a tree of tactics, and each path consists of a sequence of tactics. We initially expected that the shortening ratios would not be very high because of the selected dataset. Indeed, the Coq standard library is written by Coq experts and has been edited and improved for decades, so we expected that there is not much room to improve. However, given the size of the dataset, the proposed technique can find a number of redundant tactics, which lets us conclude that taking the after states into consideration is useful for proof shortening.

We discover many interesting cases, where proofs can be optimized. We present two examples of such proofs in Table 5. The first is about the Riemann integral where *ring* and *field* denote algebraic structures. The Coq user first substituted a subterm in the proof state, rewrote the goal by several lemmas, and finally applied a lemma about rings. However, our model discovers the non-trivial transformation on ring can be completed with a single transformation in field.

In the second example, the Coq library authors first applied the lemma `Qle_lteq` to transform the goal into a disjunction. Later, they selected the left side of the disjunction to continue the proof. Our model is able to figure out that the operation is redundant. Indeed it finds another lemma `Qlt_le_weak` that is able to immediately transform the goal to the left part of the disjunction.

In addition to such more impressive examples of simpler, shorter proofs, our model is also able to find a few abbreviations. Such abbreviations make the proof shorter but do not necessarily improve their readability. For instance, our model sometimes combines `unfold Un_growing` and `intro` into `intros x y P H n`. It uses the implicit mechanism of `intros` to unfold `Un_growing`. However, a Coq user will not be able to understand what operation `intros x y P H n` conducts without actually executing the Coq script.

Table 5. Two examples of shortening of proofs using the prediction. In both of the presented cases, a single tactic provides an equivalent transformation as a sequence of tactics. Since the hypotheses are not changed in any of the presented examples, we omit them and only present the goals for simplicity.

1	`field` makes the same transformation as (`Tactic1. Tactic2.`)
State	`1 = (x - (x + h0)) * - / h0`
Tactic1	`replace (x - (x + h0)) with (- h0); [ | ring ]`
State	`1 = - h0 * - / h0`
Tactic2	`rewrite Ropp_mult_d istr_l_reverse; rewrite Ropp_mult_distr_r_reverse; rewrite Ropp_involutive; apply Rinv_r_sym`
State	`h0 <> 0`
2	`apply Qlt_le_weak` makes the same transformation as (`Tactic1. Tactic2.`)
State	`(Qabs (xn p - yn q) <= 1 # z * k)%Q`
Tactic1	`apply Qle_lteq`
State	`(Qabs (xn p - yn q) < 1 # z * k)%Q` ∨ `(Qabs (xn p - yn q) == 1 # z * k)%Q`
Tactic2	`left`
State	`(Qabs (xn p - yn q) < 1 # z * k)%Q`

6 Related Work

Several problems originating in formal mathematics and theorem proving have been considered from the machine learning point of view. One of the most explored ones is premise selection [1]. The goal of this task is to find lemmas in a large library, that are most likely to prove a given conjecture. For premise selection, the meaning of dependency in formal mathematics has been explored using both approaches that try to explicitly define the logical semantics [19], as well as approaches that use deep learning for this [36]. Next, it is possible to apply machine learning to guide inference-based theorem provers. As part of this task, implicitly the meaning of provability and step usefulness are derived by the learning methods. This has been explored in the two top-performing first-order theorem provers [17,32] as well as in higher-order logic automated theorem proving [10]. Similarly, the meaning of the usefulness of a proof step has been considered, for example as part of the HOLStep [18], where various machine learning methods try to predict if particular inferences are needed in a proof. All these tasks are different from the task that we propose in the current paper.

Various proof automation systems have emerged to construct proofs by tactic prediction and proof search. SEPIA infers tactics for Coq by tactic trace and automata [13]. TacticToe [12] and Tactician [5,42] apply classical statistical

learning techniques such k-nearest neighbors [9] and random forests [7] to generate tactic predictions based on the before states. Several systems use neural networks for the same task, e.g. HOList [4], CoqGym [41], and Lime [40]. These are all different from the current work that considers the after states as well.

Autoformalization [20] is a machine translation task applied to formal mathematical proofs. The accuracy of the best methods applied to the task is still very weak in comparison with human formalization [37], however, the neural methods already show some minimal understanding of the meaning of formalization, for example by finding equivalent formulations. Again this is a different task from the one considered in the current work.

7 Conclusion

In this paper, we propose a new machine learning task, with which we aim to capture the semantics of tactics in formal mathematics. Based on a dataset of almost 160 thousand proof states we consider synthesizing a tactic that transforms a before state to the expected after states. We implement three novel characterizations to describe the transformation: feature difference, anti-unification, and tree difference. The results of the experiments confirm the effectiveness of our characterizations. Two applications of the task are discussed: tactic suggestion for declarative proofs and proof shortening.

In the future, we will investigate if tactic embeddings can be used directly. We can also try to estimate the after states by calculating the embeddings of the before state and the tactic or align tactics between systems in a similar way to how concepts are already aligned between systems [11].

Acknowledgements. This work was partially supported by the ERC Starting Grant *SMART* no. 714034, the ERC Consolidator grant *AI4REASON* no. 649043, the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the Cost action CA20111 EuroProofNet, the ERC-CZ project *POSTMAN* no. LL1902, Amazon Research Awards, and the EU ICT-48 2020 project TAILOR no. 952215.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reasoning **52**(2), 191–213 (2013). https://doi.org/10.1007/s10817-013-9286-5
2. Bancerek, G., et al.: The role of the Mizar mathematical library for interactive proof development in Mizar. J. Autom. Reasoning **61**, 9–32 (2018)
3. Bancerek, G., et al.: Mizar: state-of-the-art and beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 261–279. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_17
4. Bansal, K., Loos, S., Rabe, M., Szegedy, C., Wilcox, S.: HOList: an environment for machine learning of higher order logic theorem proving. In: International Conference on Machine Learning, pp. 454–463. PMLR (2019)

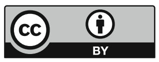
5. Blaauwbroek, L., Urban, J., Geuvers, H.: Tactic learning and proving for the Coq proof assistant. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC, vol. 73, pp. 138–150. EasyChair (2020). https://doi.org/10.29007/wg1q
6. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician. In: Benzmüller, C., Miller, B. (eds.) CICM 2020. LNCS (LNAI), vol. 12236, pp. 271–277. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_17
7. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
8. Brown, T., et al.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)
9. Dudani, S.A.: The distance-weighted k-nearest-neighbor rule. IEEE Trans. Syst. Man Cybern. **4**, 325–327 (1976)
10. Färber, M., Brown, C.: Internal guidance for Satallax. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 349–361. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_24
11. Gauthier, T., Kaliszyk, C.: Aligning concepts across proof assistant libraries. J. Symbolic Comput. **90**, 89–123 (2019). https://doi.org/10.1016/j.jsc.2018.04.005
12. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: TacticToe: Learning to prove with tactics. J. Autom. Reasoning **65**(2), 257–286 (2021)
13. Gransden, T., Walkinshaw, N., Raman, R.: SEPIA: search for proofs using inferred automata. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 246–255. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_16
14. Harrison, J.: HOL light: a tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031814
15. Harrison, J., Urban, J., Wiedijk, F.: History of interactive theorem proving. In: Computational Logic, Handbook of the History of Logic, vol. 9, pp. 135–214. Elsevier (2014)
16. Hunt, J.W., MacIlroy, M.D.: An algorithm for differential file comparison. Bell Laboratories Murray Hill (1976)
17. 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
18. Kaliszyk, C., Chollet, F., Szegedy, C.: HolStep: a machine learning dataset for higher-order logic theorem proving. In: ICLR 2017, OpenReview.net (2017)
19. Kaliszyk, C., Urban, J., Vyskocil, J.: Efficient semantic features for automated reasoning over large theories. In: Yang, Q., Wooldridge, M.J. (eds.) IJCAI 2015, pp. 3084–3090. AAAI Press (2015)
20. Kaliszyk, C., Urban, J., Vyskočil, J.: Automating formalization by statistical and semantic parsing of mathematics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 12–27. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_2
21. Miraldo, V.C., Swierstra, W.: An efficient algorithm for type-safe structural diffing. Proc. ACM Program. Lang. **3**(ICFP), 1–29 (2019)
22. Mitchell, T.M., Mitchell, T.M.: Machine Learning, vol. 1. McGraw-hill, New York (1997)
23. 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
24. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): 5. the rules of the game. In: Isabelle/HOL. LNCS, vol. 2283, pp. 67–104. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9_5
25. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions (2015)
26. Pfenning, F.: Unification and anti-unification in the calculus of constructions. In: LICS, vol. 91, pp. 74–85 (1991)
27. Plotkin, G.D.: A further note on inductive generalization. Mach. Intell. **6**, 101–124 (1971)
28. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
29. Reynolds, J.C.: Transformational systems and algebraic structure of atomic formulas. Mach. Intell. **5**, 135–151 (1970)
30. Schrittwieser, J., et al.: Mastering atari, go, chess and shogi by planning with a learned model. Nature **588**(7839), 604–609 (2020)
31. 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
32. Suda, M.: Vampire with a brain is a good ITP hammer. In: Konev, B., Reger, G. (eds.) FroCoS 2021. LNCS (LNAI), vol. 12941, pp. 192–209. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86205-3_11
33. The coq development team: Coq reference manual 8.11.1 (2020). https://coq.github.io/doc/v8.11/refman/index.html
34. Urban, J., Jakubův, J.: First neural conjecturing datasets and experiments. In: Benzmüller, C., Miller, B. (eds.) CICM 2020. LNCS (LNAI), vol. 12236, pp. 315–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_24
35. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
36. Wang, M., Tang, Y., Wang, J., Deng, J.: Premise selection for theorem proving by deep graph embedding. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
37. Wang, Q., Brown, C.E., Kaliszyk, C., Urban, J.: Exploration of neural machine translation in autoformalization of mathematics in Mizar. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020. pp. 85–98. ACM (2020). https://doi.org/10.1145/3372885.3373827
38. Wolf, T., et al.: Huggingface's transformers: state-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)
39. Wu, Y., et al.: Autoformalization with large language models. arXiv preprint arXiv:2205.12615 (2022)
40. Wu, Y., Rabe, M.N., Li, W., Ba, J., Grosse, R.B., Szegedy, C.: Lime: Learning inductive bias for primitives of mathematical reasoning. In: International Conference on Machine Learning, pp. 11251–11262. PMLR (2021)
41. Yang, K., Deng, J.: Learning to prove theorems via interacting with proof assistants. In: International Conference on Machine Learning, pp. 6984–6994. PMLR (2019)
42. Zhang, L., Blaauwbroek, L., Piotrowski, B., Černỳ, P., Kaliszyk, C., Urban, J.: Online machine learning techniques for Coq: a comparison. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) CICM 2021. LNCS (LNAI), vol. 12833, pp. 67–83. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81097-9_5

43. Zhu, Y., et al.: Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 19–27 (2015)

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.



Translating SUMO-K to Higher-Order Set Theory

Chad E. Brown[1], Adam Pease[1,2](✉), and Josef Urban[1]

[1] Czech Technical University in Prague, Prague, Czech Republic
apease@articulatesoftware.com
[2] Parallax Advanced Research, Beavercreek, OH, USA

Abstract. We describe a translation from a fragment of SUMO (SUMO-K) into higher-order set theory. The translation provides a formal semantics for portions of SUMO which are beyond first-order and which have previously only had an informal interpretation. It also for the first time embeds a large common-sense ontology into an interactive theorem proving system. We further extend our previous work in finding contradictions in SUMO from first-order constructs to include a portion of SUMO's higher-order constructs. Finally, using the translation, we can create problems that can be proven using higher-order interactive and automated theorem provers. This is tested in several systems and used to form a corpus of higher-order common-sense reasoning problems.

Keywords: ontology · theorem proving · Megalodon · theorem proving · automated theorem proving · automated reasoning · SUMO

1 Introduction and Motivation

The Suggested Upper Merged Ontology (SUMO) [15,16] is a comprehensive ontology of around 20,000 concepts and 80,000 hand-authored logical statements in a higher-order logic. It has an associated integrated development environment called Sigma [19][1] that interfaces to theorem provers such as E [22] and Vampire [12]. In previous work on translating SUMO to the TPTP [25] THF (Typed Higher-order Form) [1] format, a syntactic translation to THF was created but did not resolve many aspects of the intended higher-order semantics of SUMO.

In this work, we lay the groundwork for a new translation to a language for higher-order automated theorem provers based on expressing SUMO in higher-order set theory. We believe this will attach to SUMO a stronger set-theoretical interpretation that will allow deciding more queries and provide better intuition for avoiding contradictory formalizations. Once this is done, our plan is to train ENIGMA-style [5–8] query answering and contradiction-finding [23] AITP systems on such SUMO problems and develop autoformalization [9–11,28] methods targeting common-sense reasoning based on SUMO. We believe that this is the most viable path towards common-sense reasoning that is both trainable, but also explainable and verifiable, providing an alternative to language models which come with no formal guarantees.

[1] https://www.ontologyportal.org.

© The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 255–274, 2023.
https://doi.org/10.1007/978-3-031-43369-6_14

1.1 Related Work and Contributions

In earlier work, we described [19] how to translate SUMO to the strictly first-order language of TPTP-FOF [20] and TF0 [17,18,26]. SUMO has an extensive type structure and all relations have type restrictions on their arguments. Translation to TPTP FOF involved implementing a sorted (typed) logic axiomatically in TPTP by altering all implications in SUMO to contain type restrictions on any variables that appear.

In [21] 35 SUMO queries were converted into challenge problems for first-order automated theorem provers. In many cases, first-order ATPs can prove the corresponding problem. However, some of the queries involve aspects of SUMO that go beyond first-order representation. For example, one of the queries involves a term-level binder (κ).[2] Several of the queries also involve *row variables* (also called *sequence variables*), i.e., variables that should be instantiated with a list of terms. We discuss here several such examples to motivate the translation to higher-order set theory. We then embed SUMO into the Megalodon system, providing, to our knowledge, the first representation of a large commonsense ontology within a interactive theorem prover (ITP). We then consider the higher-order problems obtained via the translation. This provides a set of challenge problems for higher-order theorem provers that come from a different source than formalized mathematics or program verification.

The rest of the paper is organized as follows. In Sect. 2 we introduce the SUMO-K fragment of SUMO, an extension of the first-order fragment of SUMO. We also show there examples in SUMO that motivate the extensions. Section 3 describes a translation from SUMO-K into a higher-order set theory. We have constructed interactive proofs of the translated form of 23 SUMO-K queries. We describe several of the proofs in Sect. 4. From the interactive proofs we obtain 4880 ATP problems and we measure the performance of higher-order automated theorem provers on this problem set in Sect. 5. Section 6 describes the planned extensions and Sect. 7 concludes. Our code and problem set are available online.[3]

2 The SUMO-K Fragment

We define a fragment of SUMO we call SUMO-K. This extends the first-order fragment of SUMO with support for row variables, variable arity functions and relations, and the κ class formation term binder.[4] Elements of SUMO not included in SUMO-K are temporal, modal and probabilistic operations.

We start by defining SUMO-K *terms*, *spines* (lists of terms) and *formulas*. Formally, we have *standard variables* (x), *row variables* (ρ) and *constants* (c).

[2] Note that by "term-level binder" we mean a binder that yields a term. By way of constrast, $\forall$ and $\exists$ are formula-level binders. κ is used to form classes in SUMO. Informally, one can think of $\kappa x.\psi$ as the class $\{x|\psi\}$.

[3] http://grid01.ciirc.cvut.cz/~chad/sumo2set-0.9.tgz.

[4] SUMO classes should not be confused with set-theoretic classes. Our use of "class" in this paper will always refer to SUMO classes.

We will also have *signed rationals* (q) represented by a decimal expression with finitely many digits (i.e., those rationals expressible in such a way) as terms. We define by mutual recursion the sets of SUMO-K terms t, SUMO-K spines s and SUMO-K formulas ψ as follows:

$$\begin{array}{lll} t & ::= & x|c|q|(x\ s)|(c\ s)|(\kappa x.\psi)|\mathsf{Real}|\mathsf{Neg}|\mathsf{Nonneg}|(t\ +\ t)|(t\ -\ t)|(t\ *\ t)|(t\ /\ t) \\ s & ::= & t\ s|\cdot|\rho|\rho\ t \cdots\ t \\ \psi & ::= & \bot|\top|(\neg\psi)|(\psi \to \psi)|(\psi \wedge \psi)|(\psi \vee \psi)|(\psi \leftrightarrow \psi)|(\forall x.\psi)|(\exists x.\psi)|(\forall \rho.\psi)|(\exists \rho.\psi) \\ & | & (t = t)|(\texttt{instance}\ t\ t)|(\texttt{subclass}\ t\ t)|(t\ \leq\ t)|(t\ <\ t)|(c\ s) \end{array}$$

The definition is mutually recursive since the term $\kappa x.\psi$ depends on the formula ψ. Of course, κ, $\forall$ and $\exists$ are binders. In practice, most occurrences of ρ are at the end of the spine. In some cases, however, extra arguments $t_1, \ldots, t_n$ occur after the ρ. The idea is that ρ will be a list of arguments and $t_1, \ldots, t_n$ will be appended to the end of that list. Note that at most one row variable can occur in a spine.

2.1 Implicit Type Guards

Properly parsing SUMO terms and formulas requires mechanisms for inferring implicit type guards for variables (interpreted conjunctively for κ and $\exists$ and via implication for $\forall$). Free variables in SUMO assertions are implicitly universally quantified and are restricted by inferred type guards, as described in [19]. In previous translations targeting first-order logic, relation and function variables are instantiated during the translation (treating the general statement quantifying over relations and functions as a macro to be expanded). Since the current translation will leave these as variables, we must also deal with type guards that are not known until the relation or function is instantiated.

2.2 Variable Arity Relations and Functions

Consider the SUMO relation partition, declared as follows:

```
(instance partition Predicate)
(instance partition VariableArityRelation)
(domain partition 1 Class)
(domain partition 2 Class)
```

The last three items indicate that partition has variable arity with at least 2 arguments, both of which are intended to be classes. If there are more than 2 arguments, the remaining arguments are also intended to be classes. In general, the extra optional arguments of a variable arity relation or function are intended to have the same domain as the last required argument. We will translate partition to a set that encodes not only when the relation should hold, but also its domain information, its minimum arity and whether or not it is variable arity.

Two other variable arity relations (with the same arity and type information as partition) are exhaustiveDecomposition and disjointDecomposition. The following is an example of a SUMO-K assertion relating these concepts:

$$\forall \rho.\mathsf{partition}\ \rho \to \mathsf{exhaustiveDecomposition}\ \rho\ \wedge\ \mathsf{disjointDecomposition}\ \rho.$$

Previous translations to first-order logic expanded this assertion into several facts for different possible arities (using different predicates $\mathsf{partition}_3$, $\mathsf{partition}_4$, etc.), up to some limit. The following is an example of a partition occurring in Merge.kif[5] with 6 arguments:

```
(partition Word Noun Verb Adjective Adverb ParticleWord)
```

From this one should be able to infer the following query:

*Example 1 (*wordex*).*

```
(query (exhaustiveDecomposition
          Word Noun Verb Adjective Adverb ParticleWord))
```

However, the corresponding first-order problem will not be provable unless the limit on the generated arity is at least 6. Our translation into set theory will free us from the need to know such limits in advance.

2.3 Quantification over Relations

Merge.kif includes assertions that quantify over relations. The following is an example of such an assertion:

```
(=>
  (and
    (subrelation ?REL1 ?REL2)
    (instance ?REL1 Predicate)
    (instance ?REL2 Predicate)
    (?REL1 @ROW))
  (?REL2 @ROW))
```

In previous first-order translations such assertions are instantiated with all R and R' where (subrelation R R') is asserted. One of the 35 problems from [21] (TQG22) makes use of the SUMO assertion that son is a subrelation of parent and the macro expansion style of first-order translation is sufficient to handle this example. However, the macro expansion approach is insufficient to handle hypothetical subrelation assertions. The following is an example of a query creating a hypothetical subrelation assertion:

[5] Merge.kif is the main SUMO ontology file. While Merge.kif evolves over time, we work with a fixed version of the file from January 2023. Latest versions of it and all the other files that make up SUMO are available at https://github.com/ontologyportal/sumo.

*Example 2 (*TQG22alt4*).*

```
(query (=> (exists (?X) (employs ?X ?X))
           (not (subrelation employs uses))))
```

During the process of answering this query we will assume employs is a subrelation of uses and then must instantiate the general assertion about subrelations with employs and uses. Our translation to set theory will permit this.

2.4 Kappa Binders

One of the 35 queries from [21] (TQG27) has the following local assumption making use of a κ-binder.

Example 3. The example TQG27 includes three assertions:
(A1) instance Planet Class, (A2) subclass Planet AstronomicalBody, and (the one with a κ-binder) (A3) instance o (κp.instance p Planet $\wedge$ attribute p Earthlike). Informally, one can read (A3) as $o \in \{p|p$ is an Earthlike planet$\}$. The query is (Q) instance o Planet.

The query should easily follow by eliminating the κ-abstraction. The first-order problem generated in [21] drops the assumption with the κ-abstraction (A3), making the problem unlikely to be provable (at least not for the intended reason). Our translation to set theory will handle κ-binders and the translation of this problem will be provable in the set theory.

2.5 Real Arithmetic

Six of the 35 examples from [21] involve some real arithmetic. Two simple example queries are the following:

*Example 4 (*TQG3*).*

```
(instance Number3-1 NonnegativeRealNumber)
(query (not (instance Number3-1 NegativeRealNumber)))
```

*Example 5 (*TQG11*).* `(query (equal 12 (MultiplicationFn 3 4)))`

For the sake of brevity we represent the first problem as having one local constant n, one local assumption instance n Nonneg and the query (conjecture) $\neg$(instance n Neg). We will translate signed rationals with a finite decimal expansion to real numbers represented as sets.[6] We will also translate Real to be equal to the set of reals $\Re$. Furthermore we translate the operations $+$, $-$, $*$, $/$, $<$ and $\leq$ to have the appropriate meaning when applied to two reals.[7] We then translate

[6] We use a fixed construction of the reals, but the details of this are not relevant here.

[7] To be more precise, we are using a specific set of reals constructed in the higher-order set theory, and operations (e.g., multiplication) are the expected set-theoretic operations on that set of reals. For simplicity, our set-theoretic division is a total function returning 0 when the denominator is 0.

Neg to $\{x \in \Re | x < 0\}$ and Nonneg to $\{x \in \Re | 0 \leq x\}$. Using the properties of the set-theoretic encoding, the translated queries above are set-theoretic theorems.

In addition to direct uses of arithmetic as in the examples above, arithmetic is also often used to check type guard information. This is due to the fact that a spine like $t_1\, t_2\, \rho$ will use subtraction to determine that under some constraints the ith element of the corresponding list will be the $(i-2)^{nd}$ element of the list interpreting ρ.

3 Translation of SUMO-K to Set Theory

3.1 High Level Overview: Sets, Terms, Spines and Formulas

Our translation maps terms t to sets. The particular set theory we use is *higher-order Tarski-Grothendieck* as described in [4].[8] The details of this set theory are not important here. We only note that we have $\in$, $\subseteq$ (which will be used to interpret SUMO's `instance` and `subclass`) and that we have the ability to λ-abstract variables to form terms at higher types. The main types of interest are ι (the base type of sets), o (the type of propositions), $\iota \to \iota$ (the type of functions from sets to sets) and $\iota \to o$ (the type of predicates over sets).

Terms: When we say SUMO terms t are *translated to sets*, we mean they are translated to terms of type ι in the higher-order set theory.

Spines: Spines s are essentially lists of sets (of varying length). We translate them as functions that encode finite sequences. These functions are formally of the general type $\iota \to \iota$. However, we only use them when restricted to natural numbers, i.e., arguments $n \in \omega$ (where ω is the set of finite ordinals). We also maintain the invariant that the function returns the empty set on all but finitely many $n \in \omega$. An auxiliary function $\mathsf{listset} : (\iota \to \iota) \to \iota$ gives a set-theoretic representation of the list by restricting its domain to ω.[9]

Tagging, Untagging, Length: To avoid confusion with the empty set being on a list, we tag elements of lists to ensure they are nonempty. Let $\mathsf{I} : \iota \to \iota$ be such a *tagging function* (injective on the universe of sets) and $\mathsf{U} : \iota \to \iota$ be an *untagging function*. We then define $\mathsf{nil} : \iota \to \iota$ to be constantly $\emptyset$ and $\mathsf{cons} : \iota \to (\iota \to \iota) \to \iota \to \iota$ to take a set x and a list l to the function mapping 0 to $\mathsf{I}\ x$ and $i+1$ to $l\ i$ for $i \in \omega$. We also define a function $\mathsf{len} : (\iota \to \iota) \to \iota$ by $\lambda l.\{i \in \omega | l\ i \neq \emptyset\}$[10] giving us the length of the list (assuming it is a list). Informally, a spine $t_0 \cdots t_{n-1}$ is thus a function taking i to $\mathsf{I}(t'_i)$ for each $i \in \{0, \ldots, n-1\}$ where t'_i is the set-theoretic value of t_i and I the tagging function.

[8] Tarski-Grothendieck is a set theory in which there are universes modeling ZFC set theory. These set-theoretic universes should not be confused with the universe of discourse $\mathsf{Univ1}$ introduced below.

[9] We include $\mathsf{listset}$ since sometimes a list needs to be considered as a set.

[10] Note that by design this set is the finite ordinal giving the length of the list.

Formulas: The translation of a SUMO formula ψ can be thought of either as a set (which should be one of the sets 0 or 1) or as a proposition. We also sometimes coerce between type ι and o by considering the sets 0 and 1 to be sets corresponding to false and true. Let $\mathsf{P} : \iota \to o$ be $\lambda X.\emptyset \in X$ and let $\mathsf{B} : o \to \iota$ be $\lambda p.\mathsf{if}\ p\ \mathit{then}\ \mathsf{1}\ \mathit{else}\ \mathsf{0}$. We use these functions as coercions between ι and o.

3.2 Motivating Examples

Before describing the translation in more detail, we give a few more simple examples to explain various aspects of the translation and motivate our choices.

Univ1 and Kappa: Let Univ1 be a set. This set is intended to be a *universe of discourse* in which most (but not all) targets of interpretation for t will live. Specifically, we will map the SUMO-type Class to the set $\wp$ Univ1 (the power set of the universe). We take all SUMO-types except the four special cases Class, SetOrClass, Abstract and Entity to be sets in $\wp$ Univ1. Consequently, if a SUMO object is an instance of some class other than Class, SetOrClass, Abstract and Entity, we will know that the object is a member of Univ1. Due to this we choose to translate κ-binders using simple separation bounded by Univ1. Reconsidering TQG27 discussed in Sect. 2.4 we translate instance o (κp.instance p Planet $\wedge$ attribute p Earthlike) to a set-theoretic proposition[11] of the form $o \in \{p \in \mathsf{Univ1} | \cdots p \in \mathsf{PLANET}\ \wedge\ \cdots\}$ (only partially specified at the moment).[12] From this set-theoretic proposition we can easily derive $o \in \mathsf{PLANET}$ to solve the set-theoretic version of TQG27.

Variable Arity and Type Guards: As mentioned above, partition is a variable arity relation of at least arity 2 where every argument must be of SUMO-type Class. We will translate partition to a set PA containing multiple pieces of information. The behavior of PA as a relation is captured by the results one obtains by applying it to a set encoding a list of sets (via a set-theoretic operation $\mathsf{ap} : \iota \to \iota \to \iota$). We can apply an abstract function $\mathsf{arity} : \iota \to \iota$ to obtain the minimum arity of PA. We can apply an abstract predicate $\mathsf{vararity} : \iota \to o$ to encode that PA has variable arity. Likewise we can apply an abstract $\mathsf{domseq} : \iota \to \iota \to \iota$ to PA and an $i \in \omega$ to recover the intended domain of argument i of PA. These extra pieces of information are important to determine type guards in the presence of function and relation arguments.

In the specific case of partition the translation yields a set PA such that arity PA $= 2$, vararity PA is true and for $i \in \{0, 1, 2\}$, domseq PA $i = \wp$ Univ1. The value of domseq PA 2 determines the intended domain of all remaining (optional) arguments of the relation. (Note that SUMO indexes the first argument by 1 while in the set theory the first argument is indexed by 0.) The SUMO assertion

```
(partition Word Noun Verb Adjective Adverb ParticleWord)
```

[11] A set-theoretic proposition is a closed formula in the language of higher-order set theory [4].

[12] Note that the SUMO constant is Planet while its translated set-theoretic counterpart is PLANET.

translates to the set-theoretic statement[13]

$$\mathsf{P}\ (\mathsf{ap}\ \mathsf{PA}\ (\mathsf{listset}\ (\mathsf{cons}\ \mathsf{Word}\ (\mathsf{cons}\ \mathsf{Noun}\ (\mathsf{cons}\ \mathsf{Verb}\ (\mathsf{cons}\ \mathsf{Adjective}\ (\mathsf{cons}\ \mathsf{Adverb}\ (\mathsf{cons}\ \mathsf{ParticleWord}\ \mathsf{nil})))))))).$$

Recall the SUMO-K assertion

$$\forall\rho.\mathsf{partition}\ \rho \to \mathsf{exhaustiveDecomposition}\ \rho\ \wedge\ \mathsf{disjointDecomposition}\ \rho.$$

In this case the translation also generates type guards for the row variable ρ. Let PA, ED and DD be the sets corresponding to the SUMO constants partition, exhaustiveDecomposition and disjointDecomposition. Essentially, the assertion should only apply to ρ when ρ has at least length 2 and every entry is a (tagged) class. The translated set-theoretic statement (with type guards) is

$$\begin{aligned}\forall\rho:\iota\to\iota.&\mathsf{dom_of}\ (\mathsf{vararity}\ \mathsf{PA})\ (\mathsf{arity}\ \mathsf{PA})\ (\mathsf{domseq}\ \mathsf{PA})\ \rho\\ &\to \mathsf{dom_of}\ (\mathsf{vararity}\ \mathsf{ED})\ (\mathsf{arity}\ \mathsf{ED})\ (\mathsf{domseq}\ \mathsf{ED})\ \rho\\ &\to \mathsf{dom_of}\ (\mathsf{vararity}\ \mathsf{DD})\ (\mathsf{arity}\ \mathsf{DD})\ (\mathsf{domseq}\ \mathsf{DD})\ \rho\\ &\to \mathsf{P}\ (\mathsf{ap}\ \mathsf{PA}\ \rho) \to \mathsf{P}\ (\mathsf{ap}\ \mathsf{ED}\ \rho) \wedge \mathsf{P}\ (\mathsf{ap}\ \mathsf{DD}\ \rho)\end{aligned}$$

The statement above makes use of a new definition: $\mathsf{dom_of} : o \to \iota \to (\iota \to \iota) \to (\iota \to \iota) \to o$. The first argument of dom_of is a proposition encoding whether or not the function or relation is variable arity. In this case, all three of the propositions are variable arity (with the same typing information for all three). In the variable arity case $\mathsf{dom_of}\ \top\ n\ D\ \rho$ is defined to be $\mathsf{dom_of_varar}\ n\ D\ \rho$ where $\mathsf{dom_of_varar} : \iota \to (\iota \to \iota) \to (\iota \to \iota) \to o$, n is the minimum arity, D is the list of domain information and ρ is the list we are requiring to satisfy the guard. $\mathsf{dom_of_varar}\ n\ D\ \rho$ is defined to hold if the following three conditions hold:

1. $n \subseteq \mathsf{len}\ \rho$ (ρ has at least length n)
2. $\forall i \in n, \mathsf{U}\ (\rho\ i) \in D\ i$ and
3. $\forall i \in \mathsf{len}\ \rho, n \subseteq i \to \mathsf{U}\ (\rho\ i) \in D\ n$.

For fixed arity, dom_of is defined via a simpler dom_of_fixedar condition.

Another SUMO assertion about partitions is

```
(=>
  (partition ?SUPER ?SUB1 ?SUB2)
  (partition ?SUPER ?SUB2 ?SUB1))
```

In this case there are three standard (nonrow) variables needing type guards in the translation. Roughly speaking, domseq PA has the information we need, but in general we must modify it to be appropriate for variable arity relations. For this reason $\mathsf{domseqm} : \iota \to \iota \to \iota$ is defined to be

$$\lambda ri.\mathsf{if}\ \mathsf{vararity}\ r\ \mathsf{then}\ \mathsf{domseq}\ r\ (\mathsf{if}\ i \in\ \mathsf{arity}\ r\ \mathsf{then}\ i\ \mathsf{else}\ \mathsf{arity}\ r)\ \mathsf{else}\ \mathsf{domseq}\ r\ i.$$

[13] Note that we omit parentheses via the usual convention that implication is right associative, i.e., $\phi \to \psi \to \xi$ means $\phi \to (\psi \to \xi)$. Note also this is logically equivalent to $\phi \wedge \psi \to \xi$.

The translated statement is

$$\begin{aligned}\forall XYZ.X \in \mathsf{domseqm\ PA}\ 0 \to Y \in \mathsf{domseqm\ PA}\ 1 \to Z \in \mathsf{domseqm\ PA}\ 2\\ \to Z \in \mathsf{domseqm\ PA}\ 1 \to Y \in \mathsf{domseqm\ PA}\ 2\\ \to \mathsf{P}(\mathsf{ap\ PA}\ (\mathsf{cons}\ X\ (\mathsf{cons}\ Y\ (\mathsf{cons}\ Z\ \mathsf{nil}))))\\ \to \mathsf{P}(\mathsf{ap\ PA}\ (\mathsf{cons}\ X\ (\mathsf{cons}\ Z\ (\mathsf{cons}\ Y\ \mathsf{nil})))).\end{aligned}$$

A simpler translation for handling type guards in this example could avoid the use of dom_of and domseqm and instead look up the arity and typing information for partition, etc. This translation would not work in general since SUMO assertions quantify over relations, in which case the particular type guards are not known until the relation variables are instantiated. Consider the SUMO-K formula

$$\begin{aligned}\forall R_1 R_2.\forall \rho.\mathsf{subrelation}\ R_1\ R_2\ \wedge\ \mathsf{instance}\ R_1\ \mathsf{Predicate} \to \mathsf{instance}\ R_2\ \mathsf{Predicate}\\ \to R_1\ \rho \to R_2\ \rho.\end{aligned}$$

This translates to the set-theoretic proposition

$$\begin{aligned}\forall R_1 R_2 : \iota.\forall \rho : \iota \to \iota.R_1 \in \mathsf{domseqm\ SR}\ 0 \to R_2 \in \mathsf{domseqm\ SR}\ 1 \to\\ R_1 \in \mathsf{E} \to R_2 \in \mathsf{E} \to \mathsf{dom_of}\ (\mathsf{vararity}\ R_1)\ (\mathsf{arity}\ R_1)\ \rho\\ \to \mathsf{dom_of}\ (\mathsf{vararity}\ R_2)\ (\mathsf{arity}\ R_2)\ \rho\\ \to \mathsf{P}\ (\mathsf{ap\ SR}\ (\mathsf{cons}\ R_1\ (\mathsf{cons}\ R_2\ \mathsf{nil}))) \wedge R_1 \in \mathsf{PR} \to R_2 \in \mathsf{PR} \to \mathsf{P}\ (\mathsf{ap}\ R_1\ \rho)\\ \to \mathsf{P}\ (\mathsf{ap}\ R_2\ \rho)\end{aligned}$$

where E, SR and PR are the sets corresponding to the SUMO constants Entity, subrelation and Predicate. Here the type guards on ρ depend on R_1 and R_2. Two special cases are the type guards $R_i \in \mathsf{E}$ which are derived from the use of R_i as the first argument of instance.

3.3 The Translation

We now describe the translation itself. A first pass through the SUMO files given records the typing information from domain, range, domainsubclass, rangesubclass and subrelation assertions. A finite number of secondary passes determines which names will have variable arity (either due to a direct assertion or due to being inferred to be in a variable arity class).[14]

The final pass translates the assertions, and this is our focus here. Each SUMO-K assertion is a SUMO-K formula φ which may have free variables in it. Thus if we translate the SUMO-K formula φ into the set-theoretic proposition φ', then the translated assertion will be

$$\forall x_1 \cdots x_n.G_1 \to \cdots G_m \to \varphi'$$

where $x_1, \ldots, x_n$ are the free variables in φ and $G_1, \ldots, G_m$ are the type guards for these free variables. Note that some of these free variables may be for spine

[14] In practice with the current Merge.kif file, a single secondary pass suffices, but in general one might need an extra pass to climb the class hierarchy.

variables (i.e., row variables) and may have type $\iota \to \iota$. Such variables may also have type guards.

SUMO-K variables x translate to themselves where after translation x is a variable of type ι (ranging over sets). For SUMO-K constants c we choose a name c' and declare this as having type ι. Rational numbers q with a finite decimal expansion are translated to the set calculating the quotient of the base ten numerator divided by the appropriate power of 10. For example, 11.2 would be translated to the term $1 * 10^2 + 1 * 10 + 2$ divided by 10 (where 1, 2 and 10 are the usual finite ordinals and exponentiation by finite ordinals is defined by recursion). When a variable or constant is applied to a spine we translate the spine and use ap. As mentioned in Sect. 2.5 Real is translated to the set $\Re$, Neg is translated to $\{x \in \Re | x < 0\}$ and Nonneg is translated to $\{x \in \Re | 0 \leq x\}$. The other arithmetical constructs are translated to sets, but we assume special properties such as

$$\forall xy \in \Re.\mathsf{ap}\ \mathsf{ADD}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil})) = x + y,$$

$$\forall xy \in \Re.\mathsf{ap}\ \mathsf{MULT}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil})) = x \cdot y$$

and

$$\forall xy \in \Re.\mathsf{P}\ (\mathsf{ap}\ (\mathsf{LESSTHAN}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil})))) = (x < y).$$

- $(x\ s)$ translates to $(\mathsf{ap}\ x\ (\mathsf{listset}\ s'))$ where s' is the result of translating the SUMO-K spine s.
- $(c\ s)$ translates to $(\mathsf{ap}\ c'\ (\mathsf{listset}\ s'))$ where s' is the result of translating the SUMO-K spine s and c' is the chosen set as a counterpart to the SUMO-K constant c. Arithmetical operations are handled the same way.

The only remaining case for terms is κ binder terms.

- We translate $(\kappa x.\psi)$ to

$$\{x \in \mathsf{Univ1} \mid G_1 \wedge \ldots G_m \wedge \psi'\}$$

where $G_1, \ldots, G_m$ are generated type guards for x and ψ' is the result of translating the SUMO-K formula ψ to a set-theoretic proposition. Note that x ranges over Univ1.

The translations of spines is relatively straightforward, but a few points are worth mentioning.

- The SUMO-K spine $(t\ s)$ is translated to the list one gets by applying cons to $I\ t'$ onto s' where t' is the translation of t and s' is the translation of s.
- A spine variable ρ is translated to itself (a variable of type $\iota \to \iota$).
- In the case $\rho\ t_1\ \ldots\ t_n$ we translate ρ to itself (a variable of type $\iota \to \iota$) and translate each t_i to a set t'_i and return the function that returns $\rho\ j$ given $j < \mathsf{len}\ \rho$ and returns t'_i given $\mathsf{len}\ \rho\ +\ i$ (appending the two lists).
- The empty spine is translated to nil.

We consider each case of a SUMO-K formula. The usual logical operators are translated as the corresponding operators:

- $\bot$ and $\top$ translate simply to $\bot$ and $\top$.
- $(\neg\ \psi)$ translates to $\neg\psi'$ where ψ is a SUMO-K formula which translates to the set-theoretic proposition ψ'.
- $(\psi\ \rightarrow\ \xi)$ translates to $\psi' \rightarrow \xi'$ where ψ and ξ are SUMO-K formulas translate to the set-theoretic propositions ψ' and ξ'.
- $(\psi\ \leftrightarrow\ \xi)$ translates to $\psi' \leftrightarrow \xi'$ where ψ and ξ are SUMO-K formulas translate to the set-theoretic propositions ψ' and ξ'.
- Theoretically, $\psi \wedge \xi$ translates to $\psi' \wedge \xi'$. Practically speaking in SUMO-K conjunction is n-ary so it is more accurate to state that (and $\psi_1\ \cdots \psi_n$) translates to $\psi_1' \wedge \cdots \wedge \psi_n'$ where $\psi_1, \ldots, \psi_n$ are SUMO-K formulas translate to the set-theoretic propositions $\psi_1', \ldots, \psi_n'$.
- Again, theoretically $\psi \vee \xi$ translates to $\psi' \vee \xi'$. Practically, (or $\psi_1\ \cdots \psi_n$) translates to $\psi_1' \vee \cdots \vee \psi_n'$ where $\psi_1, \ldots, \psi_n$ are SUMO-K formulas translate to the set-theoretic propositions $\psi_1', \ldots, \psi_n'$.
- Theoretically, $\forall x.\psi$ translates to $\forall x.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where ψ' is the result of translating ψ and $G_1, \ldots, G_m$ are the generated type guards for x. Practically speaking, SUMO-K allows several variables to be universally quantified at once, so it is more accurate to say (forall $(x_1 \ldots x_n)$ ψ) translates to $\forall x_1 \ldots x_n.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where $x_1, \ldots, x_n$ are variables, $G_1, \ldots, G_m$ are the generated type guards for these variables and ψ' is the set-theoretic proposition obtained by translating ψ. That is, each G_i is a type guard induced by one of the variables x_j, with all the guards computed for the n variables simultaneously. While we could combine the guards into a single conjunction, we do not.
- $\forall\rho.\psi$ is translated similarly, but with type guards for the row variable ρ.
- Again, theoretically $\exists x.\psi$ translates to $\exists x.G_1 \wedge \cdots \wedge G_m \wedge \psi'$, where ψ' is the set-theoretic proposition obtained by translating the SUMO-K formula ψ, but generalized to handle quantifying multiple variables.
- $\exists\rho.\psi$ is translated similarly, but with type guards for the row variable ρ.
- $(t_1\ =\ t_2)$ translates to $t_1'\ =\ t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.

We use set membership and inclusion to interpret `instance` and `subclass`.

- (`instance` $t_1\ t_2$) translates to $t_1' \in t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.
- (`subclass` $t_1\ t_2$) translates to $t_1' \subseteq t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.

4 Interactive Proofs of Translated SUMO Queries

The motivating set of examples were the 35 example queries from [21], now expanded[15]. Six of the original examples involve temporal reasoning. We omit

[15] https://github.com/ontologyportal/sumo/tree/master/tests.

these for the moment, leaving a future translation to handle temporal and modal reasoning. 9 questions involve too many arguments for the existing first-order translation with macro expansion to work, but which are handled by our new translation. Among the remaining problems, 5 require some arithmetical reasoning, which use preexisting translations to standard first-order logic (FOF) and to an extension of first-order logic with arithmetic (TFF). For the remaining problems, the results of (at least) 5 were still not provable by the ATPs Vampire or E within a 600 s timeout.

We carefully looked at the set-theoretic translation of 13 of the problems that were too difficult for first-order provers (for any of the above reasons other than the use of temporal or modal reasoning). We either did an interactive proof or found slight modifications of the problem that could be interactively proven. The interactive proofs were done in Megalodon (the successor to the Egal system [4]). One advantage of having such a translation is the ability to attempt interactive proofs and recognize what may be missing from Merge.kif or the original query. We also did interactive proofs of 4 problems that the first-order provers could prove. We additionally included the 6 problems dealing with variable arity and row variables (e.g., Example 1). In total we have 23 SUMO-K queries translated to set-theoretic statements that have been interactively proven. We briefly describe some of the interactive proofs here.

An example with a particularly simple proof is TQG27 (Example 3), the example with a κ-binder. The assertion with the κ-binder translates to the set-theoretic proposition

$$o \in \{p \in \mathsf{Univ1} |\ p \in \mathsf{E} \wedge p \in \mathsf{domseqm\ attribute}\ 0 \wedge p \in \mathsf{Planet} \\ \wedge \mathsf{P}\ (\mathsf{ap\ attribute}\ (\mathsf{listset}\ (\mathsf{cons}\ p\ (\mathsf{cons\ Earthlike\ nil}))))\}.$$

The query translates simply to $o \in \mathsf{Planet}$.

When interactively proving the translated query in Megalodon, we are free to use statements coming from three sources: set-theoretic propositions already previously proven in Megalodon (or are axioms of Tarski-Grothendieck), propositions resulting from the translation of formulas in Merge.kif, and propositions resulting from translating formulas local to the example. In this case we only need two propositions: the translated formula local to the example given above and one known set-theoretic proposition of the form:

$$\forall X : \iota. \forall P : \iota \to o. \forall x : \iota. x \in \{x \in X | P\ x\} \to x \in X \wedge P\ x.$$

From the two propositions we easily obtain the conjunction

$$o \in \mathsf{Univ1} \wedge o \in \mathsf{E} \wedge o \in \mathsf{domseqm\ attribute}\ 0 \wedge o \in \mathsf{Planet} \\ \wedge \mathsf{P}\ (\mathsf{ap\ attribute}\ (\mathsf{listset}\ (\mathsf{cons}\ o\ (\mathsf{cons\ Earthlike\ nil})))).$$

After this first step, a series of steps eliminate the conjunctions until we have the desired conjunct $o \in \mathsf{Planet}$.

Another relatively simple example is TQG11 (Example 5) in which we must essentially prove 12 is $3 \cdot 4$. To be more precise we must prove

$$1 \cdot 10 + 2 = \mathsf{ap\ MULT}\ (\mathsf{listset}\ (\mathsf{cons}\ 3\ (\mathsf{cons}\ 4\ \mathsf{nil}))).$$

As mentioned in Sect. 3.3 the translation adds the proposition

$$\forall xy \in \Re.\mathsf{ap}\ \mathsf{MULT}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ \mathsf{nil})) = x \cdot y$$

which will be useful here. In the interactive proof, we first prove a claim that every natural number (finite ordinal) is a real number (i.e., $\omega \subseteq \Re$, which is true for the representation of the reals being used). This claim is then used to prove $3 \in \Re$ and $4 \in \Re$. This allows us to reduce the main goal to proving $1 \cdot 10 + 2 = 3 \cdot 4$. This goal is then proven by an unsurprising sequence of rewrites using equations defining the behavior of $+$ and $\cdot$ on finite ordinals. (Many details are elided here, such as the fact that there are actually two different operations $+$, one on reals and one only on finite ordinals and that they provably agree on finite ordinals.)

We next consider the proof of the translation of Example 2. The set-theoretic proposition resulting from translating the query is

$$\begin{array}{c}(\exists x.x \in \mathsf{domseqm}\ \mathsf{employs}\ 0 \wedge x \in \mathsf{domseqm}\ \mathsf{employs}\ 1\\ \wedge \mathsf{P}\ (\mathsf{ap}\ \mathsf{employs}\ (\mathsf{listset}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil})))))\\ \rightarrow \neg \mathsf{P}\ (\mathsf{SR}\ (\mathsf{listset}\ (\mathsf{cons}\ \mathsf{employs}\ (\mathsf{cons}\ \mathsf{uses}\ \mathsf{nil})))).\end{array}$$

We begin the interactive proof by proving the following sequence of claims:

1. $\mathsf{len}\ \mathsf{nil} = 0$.
2. $\forall X.\forall R : \iota \rightarrow \iota.\forall n.\mathsf{nat_p}\ n \rightarrow \mathsf{len}\ R = n \rightarrow \mathsf{len}\ (\mathsf{cons}\ X\ R) = \mathsf{ordsucc}\ n$.
3. $\forall y.\neg \mathsf{vararity}\ y \rightarrow \forall i.\mathsf{domseqm}\ y\ i = \mathsf{domseq}\ y\ i$.
4. $\forall y.\neg \mathsf{vararity}\ y \rightarrow \forall xi.x \in \mathsf{domseq}\ y\ i \rightarrow x \in \mathsf{domseqm}\ y\ i$.
5. $\forall X.\forall R : \iota \rightarrow \iota.\mathsf{cons}\ X\ R\ 0 = \mathsf{I}\ X$
6. $\forall n.\mathsf{nat_p}\ n \rightarrow \forall X.\forall R : \iota \rightarrow \iota.\mathsf{cons}\ X\ R\ (\mathsf{ordsucc}\ n) = \mathsf{R}\ n$.

We can then rewrite $\mathsf{domseqm}\ \mathsf{employs}$ into $\mathsf{domseq}\ \mathsf{employs}$. Starting the main body of the proof, we assume we have an x such that $x \in \mathsf{domseq}\ \mathsf{employs}\ 0$, $x \in \mathsf{domseq}\ \mathsf{employs}\ 1$ and $\mathsf{P}\ (\mathsf{ap}\ \mathsf{employs}\ (\mathsf{listset}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil}))))$. We further assume $\mathsf{P}\ (\mathsf{SR}\ (\mathsf{listset}\ (\mathsf{cons}\ \mathsf{employs}\ (\mathsf{cons}\ \mathsf{uses}\ \mathsf{nil})))$ and prove a contradiction. Using the translated Merge.kif type information from $\mathsf{employs}$ we can infer x is an autonomous agent and an object. Likewise we can infer $\mathsf{employs}$ is a predicate and a relation, and the same for uses. The contradiction follows from two claims: $\mathsf{P}\ (\mathsf{ap}\ \mathsf{uses}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil})))$ and $\neg\mathsf{P}\ (\mathsf{ap}\ \mathsf{uses}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil})))$.

We first prove $\mathsf{P}\ (\mathsf{ap}\ \mathsf{uses}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil})))$. We locally let ROW be $\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil})$ and use the claims above prove from $\mathsf{ROW}\ 0 = \mathsf{I}\ x$, $\mathsf{ROW}\ 1 = \mathsf{I}\ x$, $\mathsf{U}\ (\mathsf{ROW}\ 0) = x$, $\mathsf{U}\ (\mathsf{ROW}\ 1) = x$ and $\mathsf{len}\ \mathsf{ROW} = 2$. We can then essentially complete the subproof using the local assumptions

$$\mathsf{P}\ (\mathsf{ap}\ \mathsf{employs}\ (\mathsf{listset}\ (\mathsf{cons}\ x\ (\mathsf{cons}\ x\ \mathsf{nil}))))$$

and

$$\mathsf{P}\ (\mathsf{SR}\ (\mathsf{listset}\ (\mathsf{cons}\ \mathsf{employs}\ (\mathsf{cons}\ \mathsf{uses}\ \mathsf{nil}))))$$

along with the translation of the following Merge.kif formula:

```
(=>
  (and
    (subrelation ?REL1 ?REL2)
    (instance ?REL1 Predicate)
    (instance ?REL2 Predicate)
    (?REL1 @ROW))
  (?REL2 @ROW))
```

To complete the contradiction we prove $\neg \mathsf{P}$ (ap uses (cons x (cons x nil))). The three most significant Merge.kif formulas whose translated propositions are used in the subproof are:

```
(instance uses AsymmetricRelation)
(subclass AsymmetricRelation IrreflexiveRelation)
(=>
  (instance ?REL IrreflexiveRelation)
  (forall (?INST)
    (not
      (?REL ?INST ?INST))))
```

That is, Merge.kif declares that uses is an asymmetric relation, every asymmetric relation is an irreflexive relation, and that irreflexive relations have the expected property of irreflexivity.

5 ATP Problem Set

After interactively proving the 23 problems, we created TH0[16] problems restricted to the axioms used in the proof. This removes the need for the higher-order ATP to do premise selection. Additionally we used Megalodon to analyze the interactive proof to create a number of subgoal problems for ATPs – ranging from the full problem (the initial goal to be proven) to the smallest subgoals (completed by a single tactic). For example, the interactive proofs of Examples 1, 2 and 5 generate 415, 322 and 100 TH0 problems, respectively. In total analysis of the interactive proofs yields 4880 (premise-minimized) TH0 problems for ATPs. In Table 1 we give the results for several higher-order automated theorem provers (Leo-III [24], Vampire [13], Lash [3], Zipperposition [27], E [22]), given a 60 s timeout.

6 Future Work

The primary plan to extend the translation is to include temporal and modal operators. SUMO includes many modal operators including necessity, possibility,

[16] TH0 was introduced as THF0 in [2] as a core language for representing typed higher-order formulas (in the sense of Church's simple type theory) for automated theorem provers.

Table 1. Number of Subgoals Proven Automatically in 60 s

Problem	Subgoals	Zipperposition	Vampire	E	Lash	Leo-III
TQG1	50	50 (100%)	50 (100%)	50 (100%)	50 (100%)	50 (100%)
TQG3	20	20 (100%)	20 (100%)	14 (70%)	20 (100%)	8 (40%)
TQG7	195	188 (96%)	185 (95%)	180 (92%)	160 (82%)	158 (81%)
TQG9	19	19 (100%)	19 (100%)	19 (100%)	19 (100%)	19 (100%)
TQG10	112	112 (100%)	112 (100%)	100 (89%)	58 (52%)	96 (86%)
TQG11	100	76 (76%)	39 (39%)	67 (67%)	45 (45%)	13 (13%)
TQG19	37	34 (92%)	22 (59%)	20 (54%)	37 (100%)	11 (30%)
TQG20	41	34 (83%)	22 (54%)	20 (49%)	41 (100%)	13 (32%)
TQG21	207	154 (74%)	150 (72%)	143 (69%)	101 (49%)	56 (27%)
TQG22alt3	319	246 (77%)	214 (67%)	193 (61%)	197 (62%)	136 (43%)
TQG22alt4	322	251 (78%)	218 (68%)	197 (61%)	201 (62%)	142 (44%)
TQG22	315	271 (86%)	224 (71%)	212 (67%)	201 (64%)	142 (45%)
TQG23	67	61 (91%)	67 (100%)	42 (63%)	51 (76%)	38 (57%)
TQG25alt1	910	652 (72%)	526 (58%)	580 (64%)	529 (58%)	246 (27%)
TQG27	7	7 (100%)	7 (100%)	7 (100%)	7 (100%)	7 (100%)
TQG28alt1	600	428 (71%)	386 (64%)	349 (58%)	261 (44%)	213 (36%)
TQG30	4	4 (100%)	4 (100%)	3 (75%)	4 (100%)	4 (100%)
TQG33	112	82 (73%)	83 (74%)	79 (71%)	85 (76%)	36 (32%)
TQG45	162	136 (84%)	131 (81%)	128 (79%)	106 (65%)	36 (22%)
TQG46	344	258 (75%)	215 (62%)	225 (65%)	163 (47%)	144 (42%)
TQG47	186	141 (76%)	113 (61%)	109 (59%)	93 (50%)	79 (42%)
TQG48	336	249 (74%)	234 (70%)	219 (65%)	184 (55%)	146 (43%)
wordex	415	315 (76%)	255 (61%)	236 (57%)	284 (68%)	143 (34%)
Total	4880	3788 (78%)	3296 (68%)	3192 (65%)	2897 (59%)	1936 (40%)

deontological operators (obligation and permission) and modalities for knowledge, beliefs and desires. Each modality can be modelled using Kripke style semantics [14] (possible worlds with an accessibility relation).

The following is an example of a SUMO formula in Merge.kif using modalities[17] :

```
(=>
  (modalAttribute ?FORMULA Necessity)
  (modalAttribute ?FORMULA Possibility))
```

[17] Note that SUMO embeds several different modalities that have different axiomatizations. Rather than assuming one particular modal logic axiomatization (S4, S5 etc.) by embedding different modal logics in higher-order logic we hope to determine if we can create a coherent system of axiomatizations while avoiding known paradoxes like the gentle murderer paradox.

The current translation simply skips these formulas as they are not in the SUMO-K fragment. If we only wanted to extend the translation to include necessity and possibility, we could change the translation to make the dependence on worlds explicit. The SUMO formula above could translate to the proposition

$$\forall w \in W.\forall \varphi : \iota \to \iota.(\forall v \in W.R\ w\ v \to \mathsf{P}\ (\varphi\ v)) \to (\exists v \in W.R\ w\ v \wedge \mathsf{P}\ (\varphi\ v)).$$

Here W is a set of worlds and R is an accessibility relation on W. Note that the translated formula variable has type $\iota \to \iota$ instead of type ι to make the dependence of the formula on the world explicit. In general, terms, spines and formulas would depend on a world w and in an asserted formula the world w would be universally quantified (ranging over W) as above.

If we took the approach above to model necessity and possibility, then to add deontic modalities later we would need a second set of worlds and accessibility relation. The translation of terms would then have type $\iota \to \iota \to \iota$ to account for the dependence on both kinds of worlds. In order to prevent needing to keep adding new dependencies for every modalities, our plan is to combine the sets of worlds and accessibility relations in an extensible way. Thus terms will translate to have type $\iota \to \iota$ essentially giving dependence on a single set encoding a sequence of worlds (where we are open ended about the length of the sequence). Using this idea, the SUMO formula above would translate to something like

$$\forall w \in (\Pi x \in X.W\ x).\forall \varphi : \iota \to \iota.(\forall v \in (\Pi x \in X.W\ x).R\ m\ w\ v \to \mathsf{P}\ (\varphi\ v))$$
$$\to (\exists v \in (\Pi x \in X.W\ x).R\ m\ w\ v \wedge \mathsf{P}\ (\varphi\ v))$$

where X is an index set (where each $x \in X$ corresponds to a modality being interpreted), $m \in X$ is the specific index for necessity and possibility, $W\ x$ is the set of worlds for x, and $R\ x$ is a relation between $w, v \in \Pi x \in X.W\ x$ that holds if the x components satisfy the accessibility relation over $W\ x$ and the other components of w and v do not change. This allows us to model an arbitrary number of modalities using Kripke semantics while only carrying one world argument. Another advantage is that it minimizes the change to the translation of formulas in the SUMO-K fragment (without modalities). The only required change is to add a single dependence on w via a new argument and universally quantify over w if the formula is asserted.

We have already done some experiments with this approach and it shows promise. The previous experiments need to be extended to include changes that have occurred to obtain the SUMO-K translation described in the present paper. Once this is done, we must ensure that translated examples both with modalities and the examples in this paper without modalities are provable interactively. We plan to also test automated theorem provers on the subgoals obtained from the interactive proofs. Doing so with the 23 examples in this paper will give an indication how much more difficult the translated problems become if the Kripke infrastructure to handle modalities is included.

Another aspect of SUMO are modalities involving likelihood and probability. These cannot be modelled by Kripke semantics (as the modalities are not normal). We are experimenting with using neighborhood semantics to include these modalities.

7 Conclusion

We have described a translation from the SUMO-K fragment of SUMO into higher-order set theory. We have considered a number of examples that use aspects of SUMO-K that go beyond traditional first-order logic, namely variable arity functions and relations, row variables, term-level κ-binders and arithmetic. We have described a number of interactive proofs of translated queries and tested higher-order automated theorem provers on problems obtained by doing premise selection using the corresponding interactive proofs. This gives a set of problems for automated theorem provers that come from the area of "common sense reasoning," an area quite different from the more common sources of formalized mathematics and program verification. On most of the examples, higher-order automated theorem provers cannot fully automatically prove the query, but they perform reasonably well on subgoal problems extracted from the interactive proofs. This gives an indication that the full problems (assuming premise selection) are not too far out of reach for current state of the art higher-order automated theorem provers.

Acknowledgments. This work was partially supported by the ERC-CZ project POSTMAN no. LL1902, Amazon Research Awards, EU ICT-48 2020 project TAILOR no. 952215 and the European Regional Development Fund under the Czech project AI&Reasoning with identifier CZ.02.1.01/0.0/0.0/15_003/0000466.

References

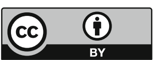
1. Benzmüller, C., Pease, A.: Higher-Order Aspects and Context in SUMO. In: Jos Lehmann, I.J.V., Bundy, A. (eds.) Special issue on Reasoning with context in the Semantic Web, vol. 12–13. Science, Services and Agents on the World Wide Web (2012)
2. Benzmüller, C., Rabe, F., Sutcliffe, G.: Thf0 - the core of the tptp language for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, pp. 491–506. Springer, Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Brown, C.E., Kaliszyk, C.: Lash 1.0 (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 350–358. Springer (2022)
4. Brown, C.E., Pąk, 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
5. Chvalovský, K., Jakubuv, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: Fontaine, P. (ed.) Automated Deduction - CADE 27–27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11716, pp. 197–215. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_12

6. Jakubuv, 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.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 448–463. Springer (2020). https://doi.org/10.1007/978-3-030-51054-1_29
7. Jakubuv, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17–21, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10383, pp. 292–302. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6
8. Jakubuv, J., Urban, J.: Hammering Mizar by learning clause guidance. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019(September), pp. 9–12, 2019. Portland, OR, USA. LIPIcs, vol. 141, pp. 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.34
9. Kaliszyk, C., Urban, J., Vyskočil, J.: Automating Formalization by Statistical and Semantic Parsing of Mathematics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 12–27. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_2
10. Kaliszyk, C., Urban, J., Vyskočil, J.: Learning to parse on aligned corpora (rough diamond). In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24–27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9236, pp. 227–233. Springer (2015). https://doi.org/10.1007/978-3-319-22102-1
11. Kaliszyk, C., Urban, J., Vyskočil, J., Geuvers, H.: Developing corpus-based translation methods between informal and formal mathematics: Project description. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7–11, 2014. Proceedings. LNCS, vol. 8543, pp. 435–439. Springer (2014). https://doi.org/10.1007/978-3-319-08434-3_34
12. 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
13. 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
14. Kripke, S.A.: Semantical analysis of modal logic i normal modal propositional calculi. Mathematical Logic Quarterly **9**, 67–96 (1963). https://doi.org/10.1002/malq.19630090502

15. Niles, I., Pease, A.: Toward a Standard Upper Ontology. In: Welty, C., Smith, B. (eds.) Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001). pp. 2–9 (2001)
16. Pease, A.: Ontology: A Practical Guide. Articulate Software Press, Angwin, CA (2011)
17. Pease, A.: Arithmetic and inference in a large theory. In: AI in Theorem Proving (2019)
18. Pease, A.: Converting the Suggested Upper Merged Ontology to Typed First-order Form arXiv:2303.04148 [cs.AI] (2023)
19. Pease, A., Schulz, S.: Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 519–525. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_40
20. Pease, A., Sutcliffe, G., Siegel, N., Trac, S.: Large Theory Reasoning with SUMO at CASC. AI Communications, Special issue on Practical Aspects of Automated Reasoning **23**(2–3), 137–144 (2010)
21. Pease, A., Sutcliffe, G., Siegel, N., Trac, S.: Large theory reasoning with sumo at casc. AI Commun. **23**(2–3), 137–144 (2010). https://doi.org/10.3233/AIC-2010-0466
22. Schulz, S.: E - A Brainiac Theorem Prover. AI Commun. **15**(2–3), 111–126 (2002)
23. Schulz, S., Sutcliffe, G., Urban, J., Pease, A.: Detecting inconsistencies in large first-order knowledge bases. In: Proceedings of CADE 26. pp. 310–325. Springer (2017)
24. Steen, A., Benzmüller, C.: The higher-order prover leo-iii. CoRR abs/1802.02732 (2018), https://arxiv.org/abs/1802.02732
25. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Proceedings of the Second International Conference on Computer Science: Theory and Applications. pp. 6–22. CSR'07, Springer-Verlag, Berlin, Heidelberg (2007), https://dl.acm.org/citation.cfm?id=2391910.2391914
26. 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
27. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28, pp. 415–432. Springer International Publishing, Cham (2021)
28. Wang, Q., Kaliszyk, C., Urban, J.: First Experiments with Neural Translation of Informal to Formal Mathematics. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) CICM 2018. LNCS (LNAI), vol. 11006, pp. 255–270. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96812-4_22

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
Al Wardani, Farah 176
Aoto, Takahito 99

B
Barrett, Clark 41, 159
Blaauwbroek, Lasse 236
Blanchette, Jasmin 23
Briefs, Yasmine 81
Bromberger, Martin 137
Brown, Chad E. 255

C
Chaudhuri, Kaustuv 176
Cignarale, Giorgio 119

D
Dahmen, Sander R. 23

E
Ekici, Burak 41

G
Giesl, Jürgen 3

H
Haga, Ryota 99
Hirokawa, Nao 63

K
Kagaya, Yuki 99
Kaliszyk, Cezary 236
Kuznets, Roman 119

L
Leidinger, Hendrik 81
Leutgeb, Lorenz 137
Lommen, Nils 3

M
Miller, Dale 176
Möhle, Sibylle 195

N
Nummelin, Visa 23

P
Pease, Adam 255

R
Rincon Galeana, Hugo 119

S
Saito, Teppei 63
Schmid, Ulrich 119

T
Tinelli, Cesare 41
Toledo, Guilherme V. 159
Torstensson, Olle 217

U
Urban, Josef 236, 255

V
Viswanathan, Arjun 41

W
Weber, Tjark 217
Weidenbach, Christoph 81, 137

Z
Zhang, Liao 236
Zohar, Yoni 41, 159

© The Editor(s) (if applicable) and The Author(s) 2023
U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, p. 275, 2023.
https://doi.org/10.1007/978-3-031-43369-6